Modal Types for Mobile Code

Tom Murphy VII

CMU-CS-08-126
May 13, 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Robert Harper, Co-Chair
Karl Crary, Co-Chair
Frank Pfenning
Peter Sewell

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: modal logic, distributed computing, programming languages, web programming, compilers, types

Abstract

In this dissertation I argue that modal type systems provide an elegant and practical means for controlling local resources in spatially distributed computer programs. A distributed program is one that executes in multiple physical or logical places. It usually does so because those places have local resources that can only be used in those locations. Such resources can include processing power, proximity to data, hardware, or the physical presence of a user. Programmers that write distributed applications therefore need to be able to reason about the places in which their programs will execute. This work provides an elegant and practical way to think about such programs in the form of a type system derived from modal logic.

Modal logic allows for reasoning about truth from multiple simultaneous perspectives. These perspectives, called "worlds," are identified with the locations in the distributed program. This enables the programming language to be simultaneously aware of the various hosts involved in a program, their local resources, and their differing perspectives on each other's code and data. This leads to a clean and general type structure for programs that respects locality while permitting high-level language features.

To argue that this system is elegant, I present a modal logic formulated for this purpose and then prove its global soundness and completeness and its equivalence to known logics. I then show how a small programming language can be derived from the logic, and how it can be implemented, proving properties of this abstract compilation procedure. All of these theorems are formalized in Twelf and can be checked by computer.

To demonstrate that it is practical, I then extend the modal calculus to a full-fledged programming language based on ML. I implemented a compiler for this language for the specific case of web applications, a distributed computation involving two hosts with widely different capabilities: the web server and the web browser. I then use the completed implementation to build realistic web applications.

Contents

List of Figures

Chapter 1

Introduction

This thesis project's goal is to demonstrate that modal type systems provide an elegant and practical means for controlling local resources in spatially distributed computer programs. To do this, I have designed a new formulation of modal logic from which I derived a modally-typed lambda calculus. I then extended this calculus to a full-fledged programming language. I implemented the programming language by building a type-directed compiler and a runtime system specialized to web applications. I then demonstrated my language's effectiveness by building realistic web applications with it. In this dissertation I present the theoretical components of my work and describe the implementation components; the source code and on-line demos are available separately.[1]

1.1 Overview

Some history will be useful to provide context for the problem I solve. This work arose from the ConCert project, in which we built theory and infrastructure for trustless peer-to-peer "Grid Computing." Grid computing, which is named by analogy with electric power grids, is the practice of coupling diverse computational resources into a large shared computer. Grid applications are mobile in the sense that they run on multiple different hosts, in different locations, during the course of their execution. One of the programming languages we implemented for the ConCert project was Grid/ML, an ML-like language with simple support for massively parallel computations.

Grid/ML. In Grid/ML, the programmer writes his whole Grid application as a unified program. This program is made of two parts: the client, which runs only on the user's computer and interacts with him, and the mobile code, which runs on arbitrary hosts in the network. (The ConCert infrastructure automatically allocates the mobile code to idle hosts.)

Grid/ML is practical for a certain class of problems, and has been used for a few Grid programming experiments. However, it has two major shortcomings. First, the

[1] They can be found at `http://tom7.org/ml5/`

mobile code is agnostic to where it is running, so applications must treat the hosts in the network uniformly; it is not possible to make use of special resources only available at certain sites. Since distributed computing is often motivated by the desire to make use of resources other than mere processing power, this rules out an important set of applications.

Second, even with the assumption of uniformity for the mobile code, Grid/ML still has a distinction between client code and mobile code. The client is the part of the Grid/ML application that the user interacts with, and the mobile code is what is actually run on the Grid. Certain operations, such as I/O, can only be performed on the client and result in run-time failure if used in mobile code.

My solution to both problems is to enrich the ML type system with a concept of *place*. By doing so, the language is able to support location-aware programming, so that network locations need not be treated uniformly. As a consequence, the client is simply another place, and we will be able to distinguish its capabilities from the capabilities of other hosts. Because we use a type system, we can make these distinctions statically, excluding errors before the program is ever run. Although the shortcomings of Grid/ML inspired ML5 (and the language would be an appropriate successor to Grid/ML), ML5 is an independent language suitable for many sorts of distributed programs. The current implementation is not designed for Grid computing with ConCert; rather, we target the Web as our application domain. This is a particular case of distributed programming where there are exactly two hosts, with widely different capabilities: the web browser and the web server.

Modal logic and a modal type system. Type systems for functional languages like ML have a close connection to logic, called the Curry-Howard isomorphism. Under this view, the propositions of intuitionistic logic are interpreted as the types of a programming language. Proofs of propositions then become programs inhabiting those types. Different logics, viewed through the lens of the Curry-Howard isomorphism, give rise to a variety of elegant and useful type systems for programming languages.

In order to develop a type system with a notion of place, we use a logic with the ability to reason spatially, namely modal logic. The propositional logic upon which ML is based is concerned with the *truth* of propositions from a single universal viewpoint. Modal logic introduces the concept of truth from multiple different perspectives, which are called "worlds." The logic is then able to reason simultaneously about truth in these worlds. Under the Curry-Howard view, these worlds become hosts in the network, and so our type system is correspondingly endowed with a notion of place.

Modal distributed programming. Because a proof in modal logic contains reasoning from multiple different worlds, programs in our modal programming language span multiple hosts. That is, each program consists of nested expressions and declarations to be evaluated at various hosts in the network. These fragments make reference to the other sites in the network, and the resources particular to those sites. The main purpose of the type system is to track these references to remote resources so that they are only

used in a safe way: Localized resources can only be used in the correct place. When code or data do not depend on their location, we are also able to indicate this in the type system. This allows them to be safely used anywhere. This brings us to the thesis statement:

> **Thesis Statement.** Modal type systems provide an elegant and practical means for controlling local resources in spatially distributed computer programs.

Although modal types naturally address spatial distribution, this is only one facet of distributed computing. In particular, a logical approach to concurrency and failure is beyond the scope of this project. However, ML5 has rudimentary support for these for the purpose of building realistic applications.

Web programming. The Web has evolved from a way of linking formatted documents together to a platform for application delivery. The modern web application is presented as a single web page with a client side program (written in JavaScript) that communicates with a server side program (written in Java or another server programming language) to access databases and other server resources. This style is known as "AJAX" and is an instance of distributed programming; the two hosts (browser and server) run in different locations and have different capabilities. For example, only the server can access the database, and only the client can interact with the user.

Implementation. My ML5 implementation targets web programming by compiling ML5 source programs into JavaScript for the client and a simple bytecode language for the server. The compiler is type-directed, meaning that it additionally transforms the types of the program through the phases of compilation, and type-checks the intermediate representations. Type-directed compilation has many conceptual and engineering benefits: it increases the robustness of the compiler by catching errors earlier, it exercises the type system in a way that informs its design, and it is necessary for our marshaling strategy.[2] Therefore we will expend a lot of effort, both in the design of the languages and in the engineering of the compiler, to make it type-directed.

The implementation also includes a web server that runs server-side code, serves the client-side code to the client, and manages the communication between the client and server. Although the runtime is specialized to web programming, the language and compiler are general enough to permit programming for an arbitrary set of hosts, including the discovery of hosts at runtime. To do this, only the runtime system would need to be extended.

[2]We also leave open the possibility of producing typed object code (certificates), but do not do that in this implementation.

1.2 Organization

The dissertation is organized following the same trajectory that the research took: an end-to-end study of programming language design and implementation from the identification of the problem domain to the crafting of applications using the completed implementation. I begin by briefly describing the ConCert project for the purpose of showing how its programming language, Grid/ML, has a type system too weak for safe distributed programming (Section 2.1). I then present our formulation of modal logic, called Lambda 5, which is the basis of a new calculus and type system for distributed programming (Section 3.2). To promote this calculus to a full-fledged programming language requires extending it in a number of ways. We first explore a classical variant, called C5, that adds continuations to the language (Section 3.4); these are used in compilation and in the implementation of threads and some programming idioms. We also find that we must extend the calculus with support for global reasoning (Section 3.5) in order to write some programs; this will also be required for compilation. I study the first few phases of compilation for our extended calculus, formalizing and proving properties in the Twelf system (Chapter 4). This concludes the foundational portion of the project.

Given these foundations, I then describe the programming language ML5 (a straightforward integration of the calculus into ML) and its implementation (Chapter 5). Following this I present my example applications, built with ML5 (Chapter 6). I conclude with a discussion of related work and ideas for the future (Chapter 7). The formalization of the calculi and proofs appear in full in the appendix.

Chapter 2

Located programming

In this chapter I motivate a manner of thinking about resources in distributed programming, which is the conceptual basis of ML5's type system. I do this by presenting the simplistic type system of Grid/ML and illustrating its shortcomings by example, and then arguing that a first-class notion of place is the appropriate way of addressing these shortcomings. This issue is not unique to Grid/ML; I also compare a few common ways that other programming languages deal with it and show that they also have unnecessary limitations. (A complete comparison to related literature appears in Section 7.1.)

2.1 ConCert and Grid/ML

For the ConCert project [16] we built a peer-to-peer infrastructure for trustless Grid computing. The ConCert system is designed to harness one specific resource: idle CPU power from a large network of volunteered computers. A program for ConCert is broken up into pieces of independent mobile code (called "cords") that may run in parallel. To the programmer, ConCert acts as a single, highly parallel computer with simple primitives for fork-join parallelism (see below).

A computer becomes part of the ConCert network by running a piece of software called the Conductor. The Conductor is responsible for maintaining contact with its peers, and for allocating cords to idle computers in order to be executed. This allocation is done using a "work-stealing" model: Each host maintains a queue of cords that are ready to be executed, and when a host is idle, it "steals" work out of the queues of other hosts. This model is efficient [9], but because the location in which a cord will eventually run depends on the idle status of the machines in the network, it is not possible for the programmer to know in advance where his code will run. Moreover, to support fault tolerance (by restarting failed cords), a cord must be able to be run multiple times in different places and always produce the same result.

As a consequence, we provide cords with a uniform view of the network—a cord cannot tell what host it is running on, nor can it access any of that host's local resources like permanent storage and I/O. This is acceptable because the only resource ConCert seeks to harness is idle CPU power. In contrast, ML5 allows programs to make use of

any sort of distributed resource.

Programmers can produce cords by hand or by using one of the programming languages we designed for ConCert, such as Grid/ML [85]. Even though cords have this uniform view, a Grid/ML program is not entirely uniform; it also includes a client part that can run only on the user's machine (because it accesses local resources like the keyboard and screen). Therefore, we still have an issue with controlling access to local resources. Let's look at an example program in Grid/ML that illustrates the language and some of the problems with it:

```
let
   val f  = openfile "numbers.txt"
   val f2 = openfile "factors.txt"

   (* append the list l to the output file *)
   fun writeresult l =
      write(f2, nums-to-string l ^ "\n")

   val inputs = readfile f

   (* prime factorization of n for n > 0 *)
   fun factor n =
      let
        fun trial 1 = n :: nil
          | trial m = if n mod m = 0
                      then factor m @ factor (n div m)
                      else trial (m - 1)
      in
        trial (floor (sqrt (real n)))
      end

   val cords =
     map (fn n => submit (fn () => factor n)) inputs

   val results = waitall cords
in
   app writeresult results
end
```

This Grid/ML program[1] computes the prime factorizations of the numbers in the file `"numbers.txt"` and writes those to the file `"factors.txt"` (both files reside on the client). It does this by creating a cord for each factoring task and submitting them to the local work queue with the primitive `submit`. The Grid/ML primitives used have these types:

[1]The example takes the liberty of assuming the existence of some library code, for instance `nums-to-string` and `readfile`.

```
submit  : (unit → α) → α cord
waitall : α cord list → α list
```

A value of type α `cord` is a running (or finished) computation that returns a result of type α. Grid/ML allows α to be instantiated with any type. In this case each factoring task is an `int list cord`. After submitting the cords, the program then waits for them all to complete, and writes the results to the output file.

The file I/O that the program performs is an effect. Such effects are allowed only in the part of the Grid/ML program that runs on the client. This is because I/O interacts with the external world, and so it depends on the place in which the code is executed. Therefore, an effect inside cord code would violate the requirement that it be agnostic about the host on which it runs. In Grid/ML, such errors are detected only at run-time. For example, if we modify the program so that the body of `factor` tries to write to the file `f2` or read from the file `f`, then the program will abort at that operation. Although both files are in scope, the file descriptors do not make sense when executing at remote sites—even if we wanted to allow I/O in cord code, we would not be able to access *those* files once execution has left the client.

This does not necessarily mean that open file descriptors can never leave the client. Suppose the program is modified to be higher-order:

```
let
   (* ... *)

   fun factor n =
      let
        (* ... *)
        val factors = trial (floor (sqrt (real n)))
      in
        (fn () => writeresult factors)
      end

   (* ... *)
in
   app (fn g => g ()) results
end
```

Now, instead of returning the list of factors directly, each cord returns a function that writes the result to the file on the client. The client consumes these results by calling the functions. This program has the same behavior as the first version. The reference to the local file safely makes a round trip from the client to the cord code and back in the environment of each function. We wish to permit programs like this one: Although this example is gratuitous, we will see many examples of useful higher-order programming in our applications (Chapter 6). Additionally, the process of compilation via CPS and closure conversion (Sections 4.5, 4.7) introduces higher-order functions and round-trip dependencies that were not evident in the source program.

2.2 Marshaling and location

To account for programs like this one, ML5 is designed around a concept of place. Each expression and variable in the program has associated with it a location. For an expression, the location indicates the place in which the expression may be evaluated to produce a value. For a bound variable, the location indicates where the value bound to it can be consumed. An important facet of this style is that it allows one part of the program—which runs on host A—to safely manipulate code and data that can only be executed or used at a different host B.

To further illustrate why this is important, let us compare a mainstay of distributed programming: the remote procedure call. Most RPC systems limit the forms of data that can be passed as arguments and returned as results from a remote procedure call. For example, CORBA limits the types of RPC arguments to a few simple base types such as strings and arrays [138, 145]. In Java RMI [37], objects matching the `Serializable` interface can be passed to remote methods, but this does not include references to local resources such as file handles. The Serializable interface is an empty "marker" interface and so can be applied to any class; it merely acts as a record of the programmer's intent that instances should be mobile. Therefore, almost no static checking is performed, and errors are caught at runtime or produce unintended behavior [72].

Java both fails to statically exclude unsafe programs (those that attempt to transmit local resources and use them remotely) and unnecessarily terminates programs that would be safe (those that transmit local resources, but do not misuse them). This is because it makes the decision at the moment the transmission is about to occur, but this is too early to know if the resource will be used in the wrong place, and too late to reject the program statically. This is a very common design in mobile programming languages, as discussed in Section 7.1.

I contend that this is a design mistake, and that it stems from a conflation of two separate notions. First is the implementation technique of marshaling, which is simply a way of representing a piece of data in a format suitable for transmission on a network. The second is the potential mobility of data; the semantic quality of making sense at more than one place. In most languages these ideas are conflated because distributed applications essentially consist of a collection of separate programs, each of which can only understand data from its own perspective. When a Java program performs a remote procedure call, the function that is called can't help but require that all its arguments make sense to it, because the only notion of "making sense" is one of "making sense locally." Therefore, every remote procedure call requires that the arguments shift from making sense to the caller to making sense at the receiver. Because this shift in perspective always occurs at the same time as marshaling, they seem like the same operation.

Located programming allows us to clearly separate these concepts by allowing us to reason from multiple simultaneous perspectives. A remote procedure call consists of two steps: preparing arguments that are appropriate for the callee, and then sending those arguments to the callee. The caller performs the first step by preparing values whose location is the callee (it can create them from scratch, use callee values that it

received previously, or convert some of its own values, subject to the rules of the type system). The type system ensures that any such values will indeed make sense to the callee. The implementation then uses its marshaling facility to transmit the values to the callee and unmarshal them. We permit any value to be marshaled, so this step can only fail if the network malfunctions. We are then able to analyze the issue of what values can be converted from one location to another as a semantic issue, not an implementation one.

We use logic to address this semantic issue in a principled way, by deriving the type system of ML5 from a spatial modal logic. Because we prove that the logic has strong soundness and completeness properties, we have good reason to believe that our type system properly embodies the relevant notions of locality and mobility—that it is not accidentally too restrictive. (In some cases we make concessions, but do so conscious of their nature.) The correspondence between the type theory and proof theory also serves as a way of recasting ideas in a different light. In this project this lead to concrete, unanticipated improvements in both the logics and programming language.

ML5 also decouples the remote call construct from the procedure (or method) construct; it simply includes a way to nest an expression to be evaluated at a remote site within a local expression. The expression may be a function call, in which case it looks like RPC; it may be some other expression, in which case the "arguments" described above are whatever free variables are used in that expression.

Local resources are an important part of distributed programming, and the located programming model gives us an effective way of controlling them. However, much of a distributed program is code or data that makes sense anywhere. For example, in the Grid/ML program above, the `factor` function refers to the functions `sqrt` and `@`, which are defined as part of the standard library, use no local resources, and can thus be used anywhere. If we required them to have locations, we would need to explicitly coerce them before using them in other locations, which would make the language very cumbersome. An important feature of the ML5 type system is that it also accounts for values like these that can be used anywhere. As a result, when ML5 is used to write non-distributed applications or when the distributed applications do not use local resources, its type system degenerates into ML's.

We have now motivated a type system enhanced with a notion of place. To summarize, the type system for ML5 will ensure that localized resources are only used in the correct place by associating with each sub-expression the place in which it will be evaluated, and associating with each bound variable the place where it may be used. We will also introduce support for bindings that are usable in any place, since this is very common. The foundation of this type system comes from modal logic, which is introduced in the next chapter.

Chapter 3

A modal logic for distributed computing

In this chapter I present the modal logics and calculi that we use as the basis of the ML5 type system. I begin with an overview of modal logic and motivate our choices for this research project in Section 3.1. In Section 3.2 I give the simplest formulation of our modal logic and lambda calculus, and then extend it with continuations in Section 3.4 and validity in Section 3.5.

3.1 Modal logic

Modal logic is actually a family of logics that share a common characteristic: They allow the simultaneous reasoning about truth from multiple different perspectives. These perspectives are called "possible worlds" (or just "worlds"). Worlds differ in the set of contingent truths that they affirm; in some worlds it is raining, in other worlds it is not.

Modal logics are distinguished by the way in which the worlds relate to one another, which is called the accessibility relation. Let us begin by using an abstract accessibility relation, writing $\mathrm{w}_1 \Rightarrow \mathrm{w}_2$ to indicate that world w_1 can access w_2 (the syntax for worlds and propositions appears in Figure 3.1).

A non-modal propositional logic, such as the one that ML is based upon, has a single universal notion of truth. That is, its principal judgment is of the form

$$\Delta \vdash A\ \mathsf{true}$$

meaning that the proposition A is true under the assumptions in Δ. Modal logic relativizes truth to worlds by instead using the judgment

$$\Gamma \vdash A\ \mathsf{true@w}$$

that is: A is true from the perspective of the world w. The context Γ collects hypotheses of the form $B\ \mathsf{true@w'}$ for various propositions B and worlds.[1] This will be the only

[1]Throughout this work we mean for Γ to support the natural structural properties such as exchange and weakening. These properties are made precise through the definition of the languages in Twelf, in the Appendix.

$$
\begin{array}{rrcl}
\text{propositions} & A, B, C & ::= & \Box A \mid A \supset B \mid \Diamond B \mid A \,\mathtt{at}\, \mathrm{w} \mid A \wedge B \mid A \vee B \mid \mathtt{p} \\
\text{primitive propositions} & \mathtt{p} & & \\
\text{world expressions} & \mathrm{w} & ::= & \mathbf{w} \mid \omega \\
\text{world variables} & \omega & & \\
\text{world constants} & \mathbf{w} & &
\end{array}
$$

Figure 3.1: Syntax of worlds and propositions. The propositions are defined by the rules of the modal logic in question; we use the same syntax in each logic for brevity. Worlds can either be named constants or variables.

notion of truth until we reintroduce universal reasoning in Section 3.5. We may also hypothesize the existence of a world (written ω world, where ω is a world variable) or its accessibility from another world (written $\mathrm{w}_1 \Rightarrow \mathrm{w}_2$).

Given this judgment, we can now prescribe the logic. We call this style of describing the logic, which is due to Simpson [126], an "explicit worlds" formulation. Some other formulations and their computational interpretations are described in Section 7.1.

The connectives $\wedge$, $\vee$, $\supset$ are essentially the same as their non-modal intuitionistic counterparts. We simply add "@w" to each of the judgments, for the same world w, to allow reasoning as usual within a particular world. (Note however that $\vee$ E allows the disjunction eliminated to come from a different world than the conclusion. This will be of concern in Section 3.2.3.) Similarly, we can only use a hypothesis $A@\mathrm{w}$ to conclude A at that same world w (Rule hyp). The modal judgment makes it possible to define new connectives $\Box$, $\Diamond$ and at. The proposition $\Box A$ means that A is true in all (accessible) worlds. If we know $\Box A@\mathrm{w}$, then we can conclude $A@\mathrm{w}'$ as long as w can access w' (Rule $\Box$ E). The accessibility condition is reflected as a premise of the rule requiring a proof of $\mathrm{w} \Rightarrow \mathrm{w}'$; the only way to prove this is to use an assumption of that form (Rule rhyp). To prove $\Box A$, we assume the existence of a hypothetical world about which nothing is known, assume we can access that world, and then prove A there (Rule $\Box$ I). If A is true at such a world, then it is true in any accessible world, because it relies on no assumptions particular to it. Reasoning at a fresh hypothetical world will be a recurring theme of this work.

The connective $\Diamond A$ means that A is true at some unknown (but accessible) world. We can prove $\Diamond A$ by giving an accessible world and a proof of A there (Rule $\Diamond$ I). If we know $\Diamond A@\mathrm{w}'$, then we can reason as follows: Assume the existence of a world ω, assume that w can access w', and assume that A is true there to conclude $C@\mathrm{w}$ (Rule $\Diamond$ E). The world variable ω stands in for the actual (now unknown) world in which A is true, just like when we eliminate an existential type, we do not know the identity of the actual type. Note that this rule involves three different worlds; this will be of concern in our computational interpretation of the logic (Section 3.2).

Following Jia and Walker [62], we also find it profitable to include one more connective not typically seen in modal logic, written $A \,\mathtt{at}\, \mathrm{w}$. With this proposition the logic belongs in the family known as "hybrid logics" [59]. Contrary to its name, I will argue

$$\frac{\Gamma \vdash A@\mathrm{w} \quad \Gamma \vdash B@\mathrm{w}}{\Gamma \vdash A \wedge B@\mathrm{w}} \wedge \mathrm{I} \qquad \frac{\Gamma \vdash A \wedge B@\mathrm{w}}{\Gamma \vdash A@\mathrm{w}} \wedge \mathrm{E}_1 \qquad \frac{\Gamma \vdash A \wedge B@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}} \wedge \mathrm{E}_2$$

$$\frac{\Gamma \vdash A@\mathrm{w}}{\Gamma \vdash A \vee B@\mathrm{w}} \vee \mathrm{I}_1 \qquad \frac{\Gamma \vdash B@\mathrm{w}}{\Gamma \vdash A \vee B@\mathrm{w}} \vee \mathrm{I}_2 \qquad \frac{\begin{array}{l}\Gamma \vdash A \vee B@\mathrm{w}' \\ \Gamma, A@\mathrm{w}' \vdash C@\mathrm{w} \quad \Gamma, B@\mathrm{w}' \vdash C@\mathrm{w}\end{array}}{\Gamma \vdash C@\mathrm{w}} \vee \mathrm{E}$$

$$\frac{\Gamma, A@\mathrm{w} \vdash B@\mathrm{w}}{\Gamma \vdash A \supset B@\mathrm{w}} \supset \mathrm{I} \qquad \frac{\Gamma \vdash A \supset B@\mathrm{w} \quad \Gamma \vdash A@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}} \supset \mathrm{E}$$

$$\frac{\Gamma \vdash A@\mathrm{w}}{\Gamma \vdash A\,\mathtt{at}\,\mathrm{w}@\mathrm{w}'} \,\mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma \vdash A\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}' \quad \Gamma, A@\mathrm{w}'' \vdash C@\mathrm{w}}{\Gamma \vdash C@\mathrm{w}} \,\mathtt{at}\ \mathrm{E}$$

$$\frac{\Gamma, \omega\ \mathsf{world}, \mathrm{w} \Rightarrow \omega \vdash A@\omega}{\Gamma \vdash \Box A@\mathrm{w}} \Box \mathrm{I} \qquad \frac{\Gamma \vdash \Box A@\mathrm{w}' \quad \Gamma \vdash \mathrm{w}' \Rightarrow \mathrm{w}}{\Gamma \vdash A@\mathrm{w}} \Box \mathrm{E}$$

$$\frac{\Gamma \vdash A@\mathrm{w}' \quad \Gamma \vdash \mathrm{w} \Rightarrow \mathrm{w}'}{\Gamma \vdash \Diamond A@\mathrm{w}} \Diamond \mathrm{I} \qquad \frac{\Gamma \vdash \Diamond A@\mathrm{w}' \quad \Gamma, \omega\ \mathsf{world}, \mathrm{w}' \Rightarrow \omega, A@\omega \vdash C@\mathrm{w}}{\Gamma \vdash C@\mathrm{w}} \Diamond \mathrm{E}$$

$$\frac{}{\Gamma, A@\mathrm{w} \vdash A@\mathrm{w}}\ \mathrm{hyp} \qquad \frac{}{\Gamma, \mathrm{w} \Rightarrow \mathrm{w}' \vdash \mathrm{w} \Rightarrow \mathrm{w}'}\ \mathrm{rhyp}$$

Figure 3.2: Intuitionistic modal logic (IK), as described by Simpson [126]. We include the "hybrid" connective `at`. Different modal logics can be produced by adding deductive rules for the relation $\Rightarrow$; here it obeys no additional structural properties. Intuitionistic S5 results when the relation is reflexive, symmetric, and transitive (Figure 3.3).

$$\frac{}{\Gamma \vdash \mathrm{w} \Rightarrow \mathrm{w}}\ \text{refl} \qquad \frac{\Gamma \vdash \mathrm{w}' \Rightarrow \mathrm{w}}{\Gamma \vdash \mathrm{w} \Rightarrow \mathrm{w}'}\ \text{sym}$$

$$\frac{\Gamma \vdash \mathrm{w} \Rightarrow \mathrm{w}' \quad \Gamma \vdash \mathrm{w}' \Rightarrow \mathrm{w}''}{\Gamma \vdash \mathrm{w} \Rightarrow \mathrm{w}''}\ \text{trans}$$

Figure 3.3: The IS5 accessibility relation

in Section 4.1 that this connective is more important and more natural than $\Box$ and $\Diamond$. It is simply an internalization of the judgment $A@\mathrm{w}$ as a proposition; to prove $A\,\mathtt{at}\,\mathrm{w}$ we prove $A@\mathrm{w}$ anywhere (Rule `at` I). To use a proof of $A\,\mathtt{at}\,\mathrm{w}''$, we introduce a hypothesis $A@\mathrm{w}''$ and go on to prove another proposition (Rule `at` E).

In our computational interpretation of modal logic, the worlds will be the hosts in the network, and proofs at those worlds will correspond to programs that can be executed at those worlds. Our next step is to choose an appropriate accessibility relation between worlds for this application.

3.1.1 Accessibility

The explicit worlds formulation of modal logic allows us to express different logics by adding deductive rules for the $\Rightarrow$ judgment. In order to justify our choice of accessibility relation, let us build intuition for the intended computational interpretation by looking at the meaning of some propositions as types:

- $A \supset B$. As in a Curry-Howard view of propositional logic, the proposition $A \supset B$ will be the type of functions from A to B.
- $\Box A$. The universal proposition $\Box A$ will be the type of code that can run anywhere (accessible) and produce a value of type A, that is, mobile code.
- $\Diamond A$. The existential proposition $\Diamond A$ will be the type of addresses pointing to an (accessible) location with a value of type A.
- $A\,\mathtt{at}\,\mathrm{w}$. The hybrid connective will be the type of an encapsulated value of type A that we know can be used specifically at w.

Since our computational model is a computer network, we take accessibility to be the ability to communicate on the network. We then choose an accessibility relation that is reflexive (a host can "access" itself, via loopback), transitive (a host can forward messages through the intermediate host) and symmetric (network connections are typically two-way). This gives us the logic IS5 (Figure 3.3). Some characteristic axioms of IS5 (given in Hilbert style; in this judgmental presentation they are simply provable propositions) illustrate why this is an appropriate choice:

- $\Box A \supset A$. If we have a piece of mobile code here, we can execute it to produce a value of type A here.
- $A \supset \Diamond A$. If we have a value of type A, then we can take its address.
- $\Box A \supset \Box\Box A$. Mobile code is itself mobile.

- $\Diamond\Diamond A \supset \Diamond A$. If we have the address of an address of a value, we can shorten this to a direct address.
- $\Diamond\Box A \supset \Box A$. If we have the address of some mobile code, we can retrieve that mobile code.
- $\Diamond A \supset \Box\Diamond A$. Addresses are mobile.
- $\Box(A \supset B) \supset \Diamond A \supset \Diamond B$. If we have a mobile function from A to B, and the address of a value of type A, then we can run that mobile function wherever the A is, and take the address of the result.

Additionally, some non-theorems illustrate the limitations on mobile code and values:

- $\not{A \supset \Box A}$. Not all values are mobile.
- $\not{\Diamond A \supset A}$. Having the address of a value does not necessarily give us that value.

Other accessibility relations have been studied in the context of distributed computing. As discussed in Section 7.2, not all real networks (including in particular the network available for web programming) are actually symmetric or transitive. IS5 is a reasonable choice because we can usually implement a symmetric and transitive network (an overlay network) atop one that is not (Section 5.5). More importantly for our purposes, IS5 admits a formulation that dispenses with the accessibility relation, leading to a more straightforward programming language. This is the topic of the next section.

3.1.2 IS5$^\cup$

The reflexive, symmetric, and transitive accessibility relation of IS5 separates worlds into a set of equivalence classes. Since every world we learn about in a derivation is related to an existing world, if we start with a single equivalence class then every world we learn about will also be in that equivalence class, and all worlds will be related. This suggests a simplification of the accessibility relation $\Rightarrow$ to the universal relation, where $\mathrm{w} \Rightarrow \mathrm{w}'$ for all w and w'. Figure 3.4 gives the simplified rules when accessibility is universal; we call the logic IS5$^\cup$.

IS5$^\cup$ and IS5 prove the same exact theorems in the empty context starting with a single constant world or set of related worlds; the only difference arises when IS5 reasons about multiple equivalence classes of worlds simultaneously. (All of the propositions we used as justification in the previous section are provable in IS5$^\cup$.) For our computational interpretation, we don't have any reason to write programs about two unconnected networks, so this distinction is unimportant to us. In fact, these two logics are so close that many just call the modal logic with universal accessibility S5 [30, 70, 116]. Regardless of its name, this is the modal logic that we base our calculus and programming language on.

$$\frac{\Gamma \vdash A@\mathrm{w} \quad \Gamma \vdash B@\mathrm{w}}{\Gamma \vdash A \wedge B@\mathrm{w}}\ \wedge \mathrm{I} \qquad \frac{\Gamma \vdash A \wedge B@\mathrm{w}}{\Gamma \vdash A@\mathrm{w}}\ \wedge \mathrm{E}_1 \qquad \frac{\Gamma \vdash A \wedge B@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}}\ \wedge \mathrm{E}_2$$

$$\frac{\Gamma \vdash A@\mathrm{w}}{\Gamma \vdash A \vee B@\mathrm{w}}\ \vee \mathrm{I}_1 \qquad \frac{\Gamma \vdash B@\mathrm{w}}{\Gamma \vdash A \vee B@\mathrm{w}}\ \vee \mathrm{I}_2 \qquad \frac{\begin{array}{l}\Gamma \vdash A \vee B@\mathrm{w}' \\ \Gamma, A@\mathrm{w}' \vdash C@\mathrm{w} \quad \Gamma, B@\mathrm{w}' \vdash C@\mathrm{w}\end{array}}{\Gamma \vdash C@\mathrm{w}}\ \vee \mathrm{E}$$

$$\frac{\Gamma, A@\mathrm{w} \vdash B@\mathrm{w}}{\Gamma \vdash A \supset B@\mathrm{w}}\ \supset \mathrm{I} \qquad \frac{\Gamma \vdash A \supset B@\mathrm{w} \quad \Gamma \vdash A@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}}\ \supset \mathrm{E}$$

$$\frac{\Gamma \vdash A@\mathrm{w}}{\Gamma \vdash A\,\mathtt{at}\,\mathrm{w}@\mathrm{w}'}\ \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma \vdash A\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}' \quad \Gamma, A@\mathrm{w}'' \vdash C@\mathrm{w}}{\Gamma \vdash C@\mathrm{w}}\ \mathtt{at}\ \mathrm{E}$$

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash A@\omega}{\Gamma \vdash \Box A@\mathrm{w}}\ \Box \mathrm{I} \qquad \frac{\Gamma \vdash \Box A@\mathrm{w}'}{\Gamma \vdash A@\mathrm{w}}\ \Box \mathrm{E}$$

$$\frac{\Gamma \vdash A@\mathrm{w}'}{\Gamma \vdash \Diamond A@\mathrm{w}}\ \Diamond \mathrm{I} \qquad \frac{\Gamma \vdash \Diamond A@\mathrm{w}' \quad \Gamma, \omega\ \mathsf{world}, A@\omega \vdash C@\mathrm{w}}{\Gamma \vdash C@\mathrm{w}}\ \Diamond \mathrm{E}$$

$$\frac{}{\Gamma, A@\mathrm{w} \vdash A@\mathrm{w}}\ \mathrm{hyp}$$

Figure 3.4: IS5$^{\cup}$. If the $\Rightarrow$ relation is the universal relation ($\mathrm{w} \Rightarrow \mathrm{w}'$ for all w and w'), then we get this simpler deductive system that does not need to track accessibility.

3.2 Lambda 5

Having chosen a logic appropriate for our problem domain of distributed computing, the next step is to add proof terms to it. These proof terms will be a computational lambda calculus whose types are the propositions of modal logic.

3.2.1 Action at a distance

Although we could give proof terms to IS5$^\cup$ directly (we get essentially the system of Jia and Walker [62]), this formulation has a quality that makes it awkward for computational purposes. Consider the rule $\Box$ E, which says that if we have a piece of mobile code (of type A) at some world w', we can evaluate it to produce a value of type A at w. Because the two worlds involved are in general different, the rule has what we call "action at a distance;" it requires coordination between two hosts to perform the operation of executing mobile code. The rules $\Diamond$ I, $\Diamond$ E, $\mathtt{at}$ I, $\mathtt{at}$ E, and $\vee$ E also have this quality. We wish for these rules, which are the primitive operations of our calculus, to have as simple behavior as possible: To analyze an object of sum type, we should not have to also evaluate some code remotely.

In this work we will eliminate action at a distance in two ways: by limiting some primitives to act only on values, and by isolating all communication between hosts into a single primitive. First, recall that the modal typing judgment $A@\mathrm{w}$ is not literally about the location of an expression or value, but about where that expression or value makes sense. In the case of an expression of type $A@\mathrm{w}$, the only thing we can do with it is run it at the world w to get a result. In the case of a value, however, we do not need to do anything because it is already a value. Consider the IS5$^\cup$ rule $\mathtt{at}$ I, augmented with proof terms:

$$\frac{\Gamma \vdash M : A@\mathrm{w}'}{\Gamma \vdash \mathtt{hold}\ M : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{I}$$

In order to evaluate an expression $\mathtt{hold}\ M$ at the world w, we must evaluate M at w', return the value to w and wrap it; this is action at a distance. However, if M is already a value, then we can perform the action locally.

The other technique is to decompose rules like $\mathtt{at}$ I into two parts: A local rule that requires its argument to be at the same world, and a single rule (called $\mathtt{get}$) that allows us to transfer control and data between worlds. It is not obvious that rewriting the rules this way does not damage the logic; the major technical content of this section is a proof that our calculus is equivalent (for provability) to IS5$^\cup$.

Because our logic does not make a distinction between value and expression, we will postpone discussion of the first point until we transition to a more computational perspective in Chapter 4. However, note that the calculus we present here is the most conservative one—every primitive acts only locally—yet it retains all of the expressiveness of IS5$^\cup$. Relaxing our locality restriction for some rules (when they act on values) therefore will not change the expressiveness of the language.

$$\frac{\Gamma \vdash M : A @ \mathrm{w} \quad \Gamma \vdash N : B @ \mathrm{w}}{\Gamma \vdash \langle M, N \rangle : A \wedge B @ \mathrm{w}} \wedge \mathrm{I} \qquad \frac{\Gamma \vdash M : A \wedge B @ \mathrm{w}}{\Gamma \vdash \#1\, M : A @ \mathrm{w}} \wedge \mathrm{E}_1 \qquad \frac{\Gamma \vdash M : A \wedge B @ \mathrm{w}}{\Gamma \vdash \#2\, M : B @ \mathrm{w}} \wedge \mathrm{E}_2$$

$$\frac{\Gamma \vdash M : A @ \mathrm{w}}{\Gamma \vdash \mathtt{inl}\, M : A \vee B @ \mathrm{w}} \vee \mathrm{I}_1 \qquad \frac{\Gamma \vdash M : B @ \mathrm{w}}{\Gamma \vdash \mathtt{inr}\, M : A \vee B @ \mathrm{w}} \vee \mathrm{I}_2$$

$$\frac{\begin{array}{l}\Gamma \vdash M : A \vee B @ \mathrm{w} \\ \Gamma, x{:}A @ \mathrm{w} \vdash N_1 : C @ \mathrm{w} \quad \Gamma, y{:}B @ \mathrm{w} \vdash N_2 : C @ \mathrm{w}\end{array}}{\begin{array}{l}\mathtt{case}\, M\, \mathtt{of} \\ \quad \mathtt{inl}\, x \Rightarrow N_1 \\ \quad \mathtt{inr}\, y \Rightarrow N_1\end{array} : C @ \mathrm{w}} \vee \mathrm{I}_1$$

$$\frac{\Gamma, x{:}A @ \mathrm{w} \vdash M : B @ \mathrm{w}}{\Gamma \vdash \lambda x.M : A \supset B @ \mathrm{w}} \supset \mathrm{I} \qquad \frac{\Gamma \vdash M : A \supset B @ \mathrm{w} \quad \Gamma \vdash N : A @ \mathrm{w}}{\Gamma \vdash M\, N : B @ \mathrm{w}} \supset \mathrm{E}$$

$$\frac{\Gamma \vdash M : A @ \mathrm{w}}{\Gamma \vdash \mathtt{hold}\, M : A\, \mathtt{at}\, \mathrm{w} @ \mathrm{w}} \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma \vdash M : A\, \mathtt{at}\, \mathrm{w}'' @ \mathrm{w} \quad \Gamma, x{:}A @ \mathrm{w}'' \vdash N : C @ \mathrm{w}}{\Gamma \vdash \mathtt{leta}\, x = M\, \mathtt{in}\, N : C @ \mathrm{w}} \mathtt{at}\ \mathrm{E}$$

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash M : A @ \omega}{\Gamma \vdash \mathtt{box}\, \omega.M : \Box A @ \mathrm{w}} \Box \mathrm{I} \qquad \frac{\Gamma \vdash M : \Box A @ \mathrm{w}}{\Gamma \vdash \mathtt{unbox}\, M : A @ \mathrm{w}} \Box \mathrm{E}$$

$$\frac{\Gamma \vdash M : A @ \mathrm{w}}{\Gamma \vdash \mathtt{here}\, M : \Diamond A @ \mathrm{w}} \Diamond \mathrm{I} \qquad \frac{\Gamma \vdash M : \Diamond A @ \mathrm{w} \quad \Gamma, \omega\ \mathsf{world}, x{:}A @ \omega \vdash N : C @ \mathrm{w}}{\Gamma \vdash \mathtt{letd}\, \omega, x = M\, \mathtt{in}\, N : C @ \mathrm{w}} \Diamond \mathrm{E}$$

$$\frac{}{\Gamma, x{:}A @ \mathrm{w} \vdash x : A @ \mathrm{w}}\ \mathrm{hyp} \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : A @ \mathrm{w}'}{\Gamma \vdash \mathtt{get}[\mathrm{w}']\, M : A @ \mathrm{w}}\ \mathrm{get}$$

Figure 3.5: Lambda 5 natural deduction, with proof terms. Compared to IS5$^\cup$, we have added the `get` rule and constrained the remainder of the rules to act locally.

3.2.2 Lambda 5 natural deduction

The natural deduction for Lambda 5 is given in Figure 3.5. We have added proof terms; the judgment is now

$$\Gamma \vdash M : A @ \mathrm{w}$$

meaning that M is a proof of $A @ \mathrm{w}$ under the hypotheses in Γ, which now contains hypotheses of the form $x_1{:}A_1 @ \mathrm{w}_1$ and ω world. (The presence of proof terms also allows us to distinguish this judgment from the similar one of IS5$^\cup$.) For the connectives $\supset$, $\wedge$, and $\vee$ the proof terms are as usual. Let us preview the computational interpretation of the others. For the `at` modality, we have `hold` M, which wraps the value of M to mark it as possibly belonging to another world. The elimination form, `leta` $x = M$ `in` N binds the variable x at that other world. (Binding a variable for a remote world is not action at a distance, because it requires no remote computation.) For $\Box$, we have `box` $\omega.M$,

which suspends the execution of the mobile code M until we use `unbox` to evaluate it. A value of type $\Diamond A$ can be introduced with `here` M, if M is of type A at the same world. When we consume a value of type $\Diamond A$ with `letd` $\omega, x = M$ `in` N we learn of a new hypothetical world ω and bind a variable of type $A@\omega$. Note that all of these rules act locally. If we want to reason across worlds, we use the `get` construct. `get`$[\mathrm{w}']\ M$ at the world w evaluates the expression M at w' and returns the resulting value to w. The evaluation of M at w' produces a value whose type is $A@\mathrm{w}'$, but `get`$[\mathrm{w}']\ M$ has type $A@\mathrm{w}$. We therefore have a restriction on the types A that we can use `get` on; without this, all worlds would conclude the same set of facts, making the logic too degenerate to be useful.

$$\frac{}{\Box A \text{ mobile}}\ \Box\,\mathrm{M} \qquad \frac{}{\Diamond A \text{ mobile}}\ \Diamond\,\mathrm{M}$$

$$\frac{}{A\ \texttt{at}\ \mathrm{w} \text{ mobile}}\ \texttt{at}\ \mathrm{M}$$

$$\frac{A \text{ mobile} \quad B \text{ mobile}}{A \wedge B \text{ mobile}}\ \wedge\,\mathrm{M} \qquad \frac{A \text{ mobile} \quad B \text{ mobile}}{A \vee B \text{ mobile}}\ \vee\,\mathrm{M}$$

Figure 3.6: Definition of the mobile judgment.

We restrict `get` to types that satisfy the mobile judgment, which is given in Figure 3.6. Thinking computationally, a type is mobile if every value of that type is portable to any world; for example, strings and integers are mobile types, as are pairs of mobile types. An encapsulated value $A\ \texttt{at}\ \mathrm{w}$ is mobile no matter what A is. In Chapter 4 we will see this property formalized for the proof of type safety.

The mobility judgment also has a logical justification, which we use in the proof that Lambda 5 is equivalent to IS5$^{\cup}$ below.

Examples. Some examples will help to illustrate the interaction between `get` and the local rules. Here are proofs of some of the propositions we used to motivate our choice of S5 in Section 3.1.1, assuming some constant world $\mathbf{w}$ at which to prove them:

1. $\vdash \lambda x.\texttt{unbox}\ x : \Box A \supset A@\mathbf{w}$
2. $\vdash \lambda x.\texttt{here}\ x : A \supset \Diamond A@\mathbf{w}$
3. $\vdash \lambda x.\texttt{letd}\ \omega, y = x\ \texttt{in}\ \texttt{get}[\omega]\ y : \Diamond\Diamond A \supset \Diamond A@\mathbf{w}$
4. $\vdash \lambda x.\texttt{letd}\ \omega, y = x\ \texttt{in}\ \texttt{get}[\omega]\ y : \Diamond\Box A \supset \Box A@\mathbf{w}$
5. $\vdash \lambda x.\texttt{box}\ \omega.\texttt{get}[\mathbf{w}]\ x : \Box A \supset \Box\Box A@\mathbf{w}$
6. $\vdash \lambda x.\texttt{box}\ \omega.\texttt{get}[\mathbf{w}]\ x : \Diamond A \supset \Box\Diamond A@\mathbf{w}$
7. $\vdash \lambda f.\lambda x.\texttt{letd}\ \omega, y = x\ \texttt{in}\ \texttt{get}[\omega]\ \texttt{here}\ ((\texttt{unbox}\ \texttt{get}[\mathbf{w}]\ f)\ a)$
   $: \Box(A \supset B) \supset \Diamond A \supset \Diamond B@\mathbf{w}$

Examples 1 and 2 are simply the eta expansions of the `unbox` and `here` primitives. Example 3 works by eliminating the outer $\Diamond$ and moving the inner one from the hypo-

$$\frac{\Gamma \Longrightarrow A@\mathrm{w} \quad \Gamma \Longrightarrow B@\mathrm{w}}{\Gamma \Longrightarrow A \wedge B@\mathrm{w}}\ \wedge \mathrm{R} \qquad \frac{\Gamma, A \wedge B@\mathrm{w}, A@\mathrm{w}, B@\mathrm{w} \Longrightarrow C@\mathrm{w}'}{\Gamma, A \wedge B@\mathrm{w} \Longrightarrow C@\mathrm{w}'}\ \wedge \mathrm{L}$$

$$\frac{\Gamma \Longrightarrow A@\mathrm{w}}{\Gamma \Longrightarrow A \vee B@\mathrm{w}}\ \vee \mathrm{R}_1 \qquad \frac{\Gamma \Longrightarrow B@\mathrm{w}}{\Gamma \Longrightarrow A \vee B@\mathrm{w}}\ \vee \mathrm{R}_2 \qquad \frac{\begin{array}{c}\Gamma, A \vee B@\mathrm{w}, A@\mathrm{w} \Longrightarrow C@\mathrm{w}' \\ \Gamma, A \vee B@\mathrm{w}, B@\mathrm{w} \Longrightarrow C@\mathrm{w}'\end{array}}{\Gamma, A \vee B@\mathrm{w} \Longrightarrow C@\mathrm{w}'}\ \vee \mathrm{L}$$

$$\frac{\Gamma, A@\mathrm{w} \Longrightarrow B@\mathrm{w}}{\Gamma \Longrightarrow A \supset B@\mathrm{w}}\ \supset\ \mathrm{R} \qquad \frac{\begin{array}{c}\Gamma, A \supset B@\mathrm{w} \Longrightarrow A@\mathrm{w} \\ \Gamma, A \supset B@\mathrm{w}, B@\mathrm{w} \Longrightarrow C@\mathrm{w}'\end{array}}{\Gamma, A \supset B@\mathrm{w} \Longrightarrow C@\mathrm{w}'}\ \supset\ \mathrm{L}$$

$$\frac{\Gamma \Longrightarrow A@\mathrm{w}}{\Gamma \Longrightarrow A\ \mathtt{at}\ \mathrm{w}@\mathrm{w}'}\ \mathtt{at}\ \mathrm{R} \qquad \frac{\Gamma, A\ \mathtt{at}\ \mathrm{w}@\mathrm{w}', A@\mathrm{w} \Longrightarrow C@\mathrm{w}''}{\Gamma, A\ \mathtt{at}\ \mathrm{w}@\mathrm{w}' \Longrightarrow C@\mathrm{w}''}\ \mathtt{at}\ \mathrm{L}$$

$$\frac{\Gamma, \omega\ \mathsf{world} \Longrightarrow A@\omega}{\Gamma \Longrightarrow \Box A@\mathrm{w}}\ \Box\ \mathrm{R} \qquad \frac{\Gamma, \Box A@\mathrm{w}, A@\mathrm{w}' \Longrightarrow C@\mathrm{w}''}{\Gamma, \Box A@\mathrm{w} \Longrightarrow C@\mathrm{w}''}\ \Box\ \mathrm{L}$$

$$\frac{\Gamma \Longrightarrow A@\mathrm{w}'}{\Gamma \Longrightarrow \Diamond A@\mathrm{w}}\ \Diamond\ \mathrm{R} \qquad \frac{\Gamma, \Diamond A@\mathrm{w}, \omega\ \mathsf{world}, A@\omega \Longrightarrow C@\mathrm{w}'}{\Gamma, \Diamond A@\mathrm{w} \Longrightarrow C@\mathrm{w}'}\ \Diamond\ \mathrm{L}$$

$$\frac{}{\Gamma, A@\mathrm{w} \Longrightarrow A@\mathrm{w}}\ \mathrm{init}$$

Figure 3.7: IS5$^{\cup}$ sequent calculus. The sequent calculus is given in terms of left and right rules instead of introduction and elimination. It has the subformula property and a cut principle (Theorem 2).

thetical world ω to $\mathbf{w}$ using `get`. Since `get` works on any mobile type, the same term has type $\Diamond\Box A \supset \Box A$ as well (Example 4). Examples 5 and 6 work by constructing box and moving the argument *to* that hypothetical world. Example 7 is the most complex. It takes a mobile function $f{:}\Box(A \supset B)@\mathbf{w}$ and the address of an argument $x{:}\Diamond A@\mathbf{w}$. Deconstructing the address gives us a hypothetical world ω and the argument at that world $y{:}A@\omega$. We travel to that world with `get`; in order to call the function, we must `get` it from our original world $\mathbf{w}$ and `unbox` it. After applying it to a, we take the address of the result with `here` and return that to $\mathbf{w}$. Note that in this example we make two round trips: one to go to ω, and one to get the mobile function from $\mathbf{w}$. We will be able to write a more direct program with a single round trip once we introduce validity in Section 3.5.

3.2.3 Soundness and completeness

To prove that Lambda 5 is equivalent to IS5$^\cup$, we will prove that it is sound and complete relative to a sequent calculus that is itself equivalent to IS5$^\cup$ (Figure 3.8). This sequent calculus appears in Figure 3.7.

The central judgment of the sequent calculus is $\Gamma \Longrightarrow A@\mathrm{w}$, where Γ is a series of hypotheses of the form $B_i@\mathrm{w}_i$. However, the rules are given in terms of left and right rules instead of introduction and elimination. Every rule (other than `init`) works by breaking down a single formula, either on the left or right side, into components. This is known as the subformula property. Due to this property, it is easy to see what sequent calculus proofs are possible for a formula. For example, any proof of $A \wedge B@\mathrm{w}$ with no hypotheses must begin with the rule $\wedge$ R, and thus contain proofs of $A@\mathrm{w}$ and $B@\mathrm{w}$ as subterms. We can also easily refute the existence of proofs, which give us consistency easily:

Theorem 1 (Consistency of IS5$^\cup$ sequent calculus)
Not all sequents are provable.

Proof: Immediate, by counterexample. Consider the sequent $\cdot \Longrightarrow \mathtt{p}@\mathbf{w}$ for some primitive proposition $\mathtt{p}$ and constant world $\mathbf{w}$. The only rule that can conclude $\mathtt{p}@\mathbf{w}$ is init, but the rule does not apply because there are no hypotheses. Therefore, it has no proof. □

In the natural deduction, there are many rules that might apply, such as $\supset$ E and $\vee$ E; the sequent calculus insists that no detours through unrelated propositions are allowed. The soundness of Lambda 5 relative to IS5$^\cup$ sequent calculus will give us consistency of Lambda 5 as well.

Soundness

In order to prove soundness, we will need a few lemmas. We will use $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$ to stand for derivations, and the syntax

$$\mathcal{D} :: \mathsf{J} \qquad \text{or} \qquad \begin{array}{c}\mathcal{D}\\ \vdots \\ \mathsf{J}\end{array}$$

to mean that $\mathcal{D}$ is a derivation of the judgment J. The first lemma is cut:

Theorem 2 (Cut)

$$\begin{array}{rl} \text{If} & \mathcal{D} :: \Gamma, \Gamma' \Longrightarrow A@\mathrm{w} \\ \text{and} & \mathcal{E} :: \Gamma, A@\mathrm{w}, \Gamma' \Longrightarrow B@\mathrm{w}' \\ \text{then} & \mathcal{F} :: \Gamma, \Gamma' \Longrightarrow B@\mathrm{w}' \end{array}$$

Cut says that if we have a proof of a proposition, we are licensed to use it as a hypothesis. The proof proceeds by lexicographic induction on (in order) the cut formula A, the derivation $\mathcal{D}$, and the derivation $\mathcal{E}$, following Pfenning [104]. It is a straightforward extension of our earlier proof [90] to a more complete set of connectives, so I only give a few cases here. All of the inductive cases of the theorem take the form of *commutative* cases or *principal* cases. A principal case for some connective is when $\mathcal{D}$ is a derivation

of the formula A by a right rule for that connective, and $\mathcal{E}$ is a use of the formula A by a left rule for that connective. Any other case is a commutative case. For example, the principal case for $A\,\mathtt{at}\,\mathrm{w}''$ is as follows; suppose

$$\mathcal{D} \;=\; \dfrac{\begin{array}{c}\mathcal{D}'\\ \vdots\\ \Gamma, \Gamma' \Longrightarrow A@\mathrm{w}''\end{array}}{\Gamma, \Gamma' \Longrightarrow A\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}}\;\mathtt{at}\;\mathrm{R} \qquad \mathcal{E} \;=\; \dfrac{\begin{array}{c}\mathcal{E}'\\ \vdots\\ \Gamma, A\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}, A@\mathrm{w}'', \Gamma' \Longrightarrow B@\mathrm{w}'\end{array}}{\Gamma, A\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}, \Gamma' \Longrightarrow B@\mathrm{w}'}\;\mathtt{at}\;\mathrm{L}$$

then we first remove the hypothesis $A\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}$ from $\mathcal{E}'$ by $\mathrm{cut}(\mathcal{D}, \mathcal{E}')$ to get

$$\mathcal{E}_2 :: \Gamma, A@\mathrm{w}'', \Gamma' \Longrightarrow B@\mathrm{w}'$$

and then remove the hypothesis $A@\mathrm{w}''$ from $\mathcal{E}_2$ with $\mathrm{cut}(\mathcal{D}', \mathcal{E}_2)$ to get

$$\mathcal{F} :: \Gamma, \Gamma' \Longrightarrow B@\mathrm{w}'$$

as required. The first appeal to induction is justified by the smaller input derivation $\mathcal{E}'$, and the second by the smaller cut formula A. Appendix A.1 contains the entire proof in machine checkable form as the relation `cut`. □

Cut is a standard property of a sequent calculus. The other lemmas are particular to our modal setting, forming the logical justification for the mobile judgment:

Theorem 3 (Expansion)
If A mobile
then $A@\mathrm{w} \Longrightarrow A@\mathrm{w}'$

Theorem 4 (Shift)
If A mobile
and $\Gamma \Longrightarrow A@\mathrm{w}$
then $\Gamma \Longrightarrow A@\mathrm{w}'$

Expansion says that we are licensed to use a hypothesis of mobile type to form the same conclusion at a different world. Shift says that if we can conclude a mobile type at one world, we can conclude it at any other world. Note that expansion is an instance of shift when Γ is $A@\mathrm{w}$ and the first derivation is the init rule, but the proof of shift appeals to expansion in precisely this case.

The proof of shift is by induction over the derivation of the sequent premise. Suppose the input derivation is of the form

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdots & \vdots\\ \Gamma \Longrightarrow A@\mathrm{w} & \Gamma \Longrightarrow B@\mathrm{w}\end{array}}{\Gamma \Longrightarrow A \wedge B@\mathrm{w}}\;\wedge\mathrm{R}$$

then by inversion on the derivation of $A \wedge B$ mobile, we have A mobile and B mobile. Therefore by induction hypothesis we have

$$\mathcal{E}_1 :: \Gamma \Longrightarrow A@\mathrm{w}'$$
$$\mathcal{E}_2 :: \Gamma \Longrightarrow B@\mathrm{w}'$$

and therefore

$$\dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \vdots & \vdots \\ \Gamma \Longrightarrow A@\mathrm{w}' & \Gamma \Longrightarrow B@\mathrm{w}'\end{array}}{\Gamma \Longrightarrow A \wedge B@\mathrm{w}'}\ \wedge \mathrm{R}$$

The cases introducing the primitively mobile types ($\Box A$, $\Diamond A$, $A\,\mathtt{at}\,\mathrm{w}$) are non-inductive. For example, if the derivation is of the form

$$\dfrac{\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \Longrightarrow A@\mathrm{w}''\end{array}}{\Gamma \Longrightarrow A\,\mathtt{at}\,\mathrm{w}@\mathrm{w}}\ \mathtt{at}\ \mathrm{R}$$

then the result is simply the derivation

$$\dfrac{\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \Longrightarrow A@\mathrm{w}''\end{array}}{\Gamma \Longrightarrow A\,\mathtt{at}\,\mathrm{w}@\mathrm{w}'}\ \mathtt{at}\ \mathrm{R}$$

because `at` R allows us to reason non-locally. (This is the reason that $A\,\mathtt{at}\,\mathrm{w}$ is mobile for any A.) The other cases are similar, except for the init case, in which we appeal directly to expansion. The full proof appears in machine checkable form in Appendix A.1 as the relation `shift`. □

The proof of expansion is a bit more interesting, proceeding by induction on the derivation of A mobile. We will give all of the cases:

rule concluding A mobile	proof of $A@\mathrm{w} \Longrightarrow A@\mathrm{w}'$
$\Box$ M	$\dfrac{\dfrac{\dfrac{}{\Box A'@\mathrm{w}, \omega\ \mathsf{world}, A@\omega \Longrightarrow A'@\omega}\ \text{init}}{\Box A'@\mathrm{w}, \omega\ \mathsf{world} \Longrightarrow A'@\omega}\ \Box\,\mathrm{L}}{\Box A'@\mathrm{w} \Longrightarrow \Box A'@\mathrm{w}'}\ \Box\,\mathrm{R}$
$\Diamond$ M	$\dfrac{\dfrac{\dfrac{}{\Diamond A'@\mathrm{w}, \omega\ \mathsf{world}, A'@\omega \Longrightarrow A'@\omega}\ \text{init}}{\Diamond A'@\mathrm{w}, \omega\ \mathsf{world}, A'@\omega \Longrightarrow \Diamond A'@\mathrm{w}'}\ \Diamond\,\mathrm{R}}{\Diamond A'@\mathrm{w} \Longrightarrow \Diamond A'@\mathrm{w}'}\ \Diamond\,\mathrm{L}$
`at` M	$\dfrac{\dfrac{\dfrac{}{A'\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}, A'@\mathrm{w}'' \Longrightarrow A'@\mathrm{w}''}\ \text{init}}{A'\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}, A'@\mathrm{w}'' \Longrightarrow A'\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}'}\ \mathtt{at}\ \mathrm{R}}{A'\,\mathtt{at}\,\mathrm{w}''@\mathrm{w} \Longrightarrow A'\,\mathtt{at}\,\mathrm{w}''@\mathrm{w}'}\ \mathtt{at}\ \mathrm{L}$

The cases for $\wedge$ M :: $A_1 \wedge A_2$ mobile and $\vee$ M :: $A_1 \vee A_2$ mobile are inductive. In each case we have subderivations of A_1 mobile and A_2 mobile. Therefore we have by

induction hypothesis $\mathcal{D}_1 :: A_1@\mathtt{w} \Longrightarrow A_1@\mathtt{w}'$ and $\mathcal{D}_2 :: A_2@\mathtt{w} \Longrightarrow A_2@\mathtt{w}'$. For $A_1 \wedge A_2$, the derivation is

$$\dfrac{\dfrac{\begin{array}{c}\text{weaken } \mathcal{D}_1 \\ \vdots \\ A_1 \wedge A_2@\mathtt{w}, A_1@\mathtt{w}, A_2@\mathtt{w} \Longrightarrow A_1@\mathtt{w}'\end{array} \qquad \begin{array}{c}\text{weaken } \mathcal{D}_2 \\ \vdots \\ A_1 \wedge A_2@\mathtt{w}, A_1@\mathtt{w}, A_2@\mathtt{w} \Longrightarrow A_2@\mathtt{w}'\end{array}}{A_1 \wedge A_2@\mathtt{w}, A_1@\mathtt{w}, A_2@\mathtt{w} \Longrightarrow A_1 \wedge A_2@\mathtt{w}'}\ \wedge \mathrm{R}}{A_1 \wedge A_2@\mathtt{w} \Longrightarrow A_1 \wedge A_2@\mathtt{w}'}\ \wedge \mathrm{L}$$

and for $A_1 \vee A_2$ it is

$$\dfrac{\dfrac{\begin{array}{c}\text{weaken } \mathcal{D}_1 \\ \vdots \\ A_1 \vee A_2@\mathtt{w}, A_1@\mathtt{w} \Longrightarrow A_1@\mathtt{w}'\end{array}}{A_1 \vee A_2@\mathtt{w}, A_1@\mathtt{w} \Longrightarrow A_1 \vee A_2@\mathtt{w}'}\ \vee \mathrm{R1} \qquad \dfrac{\begin{array}{c}\text{weaken } \mathcal{D}_2 \\ \vdots \\ A_1 \vee A_2@\mathtt{w}, A_2@\mathtt{w} \Longrightarrow A_2@\mathtt{w}'\end{array}}{A_1 \vee A_2@\mathtt{w}, A_2@\mathtt{w} \Longrightarrow A_1 \vee A_2@\mathtt{w}'}\ \vee \mathrm{R2}}{A_1 \vee A_2@\mathtt{w} \Longrightarrow A_1 \vee A_2@\mathtt{w}'}\ \vee \mathrm{L}$$

which concludes the proof. □

Note that we have some flexibility in how we define the mobile judgment. The mobility of $\Box A$, $\Diamond A$ and $A\ \mathtt{at}\ \mathtt{w}$ is required for completeness; otherwise our restriction to local introduction and elimination forms is too severe. However, the rule $\wedge$ M is not needed anywhere; we can remove it from our natural deduction and retain soundness and completeness (in fact, normalizing a natural deduction proof by applying Theorems 5 and 6 eliminates uses of $\wedge$ M). We choose to include it because it is natural to send aggregations of mobile data in distributed programs, and easy to implement. We could choose to have the rule

$$\dfrac{A\ \mathsf{mobile} \quad B\ \mathsf{mobile}}{A \supset B\ \mathsf{mobile}}\ \supset\ \mathrm{M}$$

but doing so requires a more complicated dynamic semantics (the dynamic creation of mobile proxies for functions). Therefore, we continue to make choices about the formulation of the logic that are influenced by its intended computational interpretation, but do so in a way that does not affect its underlying meaning.

Given cut and shift, we can state and prove soundness:

Theorem 5 (Soundness of Lambda 5 relative to IS5$^\cup$ sequent calculus)
If $\mathcal{D} :: \Gamma \vdash M : A@\mathtt{w}$
then $\mathcal{E} :: \ulcorner\Gamma\urcorner \Longrightarrow A@\mathtt{w}$

The operation $\ulcorner x_1 : A_1@\mathtt{w}_1, \ldots x_n : A_n@\mathtt{w}_n\urcorner$ erases variables (since they are not used in the sequent calculus), producing $A_1@\mathtt{w}_1, \ldots, A_n@\mathtt{w}_n$. Soundness simply states that for every Lambda 5 natural deduction proof (ignoring the proof terms) there exists a sequent derivation of the same formula, under the same hypotheses.

This proof is by induction over $\mathcal{D}$. The cases where $\mathcal{D}$ is an introduction rule are trivial by induction hypothesis (the right rules in the sequent calculus match the introduction rules in natural deduction). The translation of elimination rules appeals to cut; for example, if

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma \vdash \_ : \Box A@\mathtt{w}\end{array}}{\Gamma \vdash \_ : A@\mathtt{w}}\ \Box\ \mathrm{E}$$

then we have by induction hypothesis on $\mathcal{D}'$

$$\mathcal{E}_1 :: \ulcorner\Gamma\urcorner \Longrightarrow \Box A@\mathrm{w}$$

and by construction

$$\mathcal{E}_2 \quad = \quad \dfrac{\dfrac{}{\ulcorner\Gamma\urcorner, \Box A@\mathrm{w}, A@\mathrm{w} \Longrightarrow A@\mathrm{w}}\ \text{init}}{\ulcorner\Gamma\urcorner, \Box A@\mathrm{w} \Longrightarrow A@\mathrm{w}}\ \Box\,\mathrm{L}$$

so by cut$(\mathcal{E}_1, \mathcal{E}_2)$ we have

$$\mathcal{E} :: \ulcorner\Gamma\urcorner \Longrightarrow A@\mathrm{w}$$

as required. The other left rules are similar.

The only other case is `get`, where we have

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{cc}\mathcal{F}_1 & \mathcal{D}_1 \\ \vdots & \vdots \\ A\ \mathsf{mobile} & \Gamma \vdash \_ : A@\mathrm{w}'\end{array}}{\Gamma \vdash \_ : A@\mathrm{w}}\ \text{get}$$

By induction hypothesis we have

$$\mathcal{E}' :: \ulcorner\Gamma\urcorner \Longrightarrow A@\mathrm{w}'$$

and since A is mobile, by shift$(\mathcal{F}_1, \mathcal{E}')$ we have

$$\mathcal{E} :: \ulcorner\Gamma\urcorner \Longrightarrow A@\mathrm{w}$$

as required. The full proof appears in Appendix A.1 as the relation `ndseq`. □

Completeness

To show that we have not made Lambda 5 too restrictive, we prove a completeness theorem.

Theorem 6 (Completeness of Lambda 5 relative to IS5$^\cup$ sequent calculus)

If $\mathcal{D} :: \Gamma \Longrightarrow A@\mathrm{w}$
then $\mathcal{E} :: \Gamma' \vdash M : A@\mathrm{w}$, for some M and $\Gamma' \sqsupset \Gamma$

Here $\Gamma' \sqsupset \Gamma$ if Γ' has the same hypotheses as Γ, each given a unique variable name. For this proof we will need the following substitution lemmas:

Theorem 7 (World substitution for Lambda 5)
If $\Gamma \vdash \mathrm{w}\ \mathsf{world}$
and $\Gamma, \omega\ \mathsf{world}, \Gamma' \vdash M : A@\mathrm{w}'$
then $\Gamma, \Gamma' \vdash [{}^{\mathrm{w}}/_{\omega}]M : [{}^{\mathrm{w}}/_{\omega}]A@[{}^{\mathrm{w}}/_{\omega}]\mathrm{w}'$

Theorem 8 (Term substitution for Lambda 5)
If $\Gamma \vdash M : A@\mathrm{w}$
and $\Gamma, x{:}A@\mathrm{w}, \Gamma' \vdash N : C@\mathrm{w}'$
then $\Gamma, \Gamma' \vdash [{}^{M}/_{x}]N : A@\mathrm{w}'$

The world substitution operation $[{}^{\mathrm{w}}/_{\omega}]$ is defined in the obvious way on proof terms, types, world expressions, and derivations. The term substitution operation $[{}^{M}/_{x}]$ is similarly defined on proof terms. Each substitution proof is by induction on the second premise. □

Completeness is proved by induction on $\mathcal{D}$. The right rules proceed directly by induction hypothesis (since the introduction rules take the same form), except that we may insert uses of get. For example, if

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma \Longrightarrow A@\mathrm{w}'\end{array}}{\Gamma \Longrightarrow A\,\mathtt{at}\,\mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{R}$$

then we have by induction hypothesis on $\mathcal{D}'$

$$\mathcal{E}' :: \Gamma' \vdash M : A@\mathrm{w}'$$

which is at the wrong world to use with the local-only Lambda 5 rule `at` I, but by using get we have

$$\mathcal{E} \quad = \quad \dfrac{\dfrac{}{A\ \mathsf{mobile}}\ \mathtt{at}\ \mathrm{M} \quad \dfrac{\begin{array}{c}\mathcal{E}' \\ \vdots \\ \Gamma' \vdash M : A@\mathrm{w}'\end{array}}{\Gamma' \vdash M : A\,\mathtt{at}\,\mathrm{w}'@\mathrm{w}'}\ \mathtt{at}\ \mathrm{I}}{\Gamma' \vdash \mathtt{get}[\mathrm{w}']\ M : A\,\mathtt{at}\,\mathrm{w}'@\mathrm{w}}\ \mathrm{get}$$

as required.

The translations of left rules appeal to substitution. For example, if

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma, \Box A@\mathrm{w}, A@\mathrm{w}' \Longrightarrow C@\mathrm{w}''\end{array}}{\Gamma, \Box A@\mathrm{w} \Longrightarrow C@\mathrm{w}''}\ \Box\ \mathrm{L}$$

then by induction hypothesis we have

$$\mathcal{E}' \quad = \quad \Gamma', x{:}\Box A@\mathrm{w}, y{:}A@\mathrm{w}' \vdash M : C@\mathrm{w}''$$

and we can discharge the hypothesis by substitution. Let

$$\mathcal{F} \quad = \quad \dfrac{\dfrac{\dfrac{}{\Box A\ \mathsf{mobile}}\ \Box\ \mathrm{M} \quad \dfrac{}{\Gamma, x{:}\Box A@\mathrm{w} \vdash x : \Box A@\mathrm{w}}\ \mathrm{hyp}}{\Gamma, x{:}\Box A@\mathrm{w} \vdash \mathtt{get}[\mathrm{w}]\ x : \Box A@\mathrm{w}'}\ \mathrm{get}}{\Gamma, x{:}\Box A@\mathrm{w} \vdash \mathtt{unbox}\ \mathtt{get}[\mathrm{w}]\ x : A@\mathrm{w}'}\ \Box\ \mathrm{E}$$

and then by term substitution on $\mathcal{F}$ and $\mathcal{E}'$ we have

$$\mathcal{E} :: \Gamma', x{:}\Box A@\mathrm{w} \vdash [{}^{\mathtt{unbox\ get}[\mathrm{w}]\ x}/_{y}]M : C@\mathrm{w}''$$

as required.

Disjunction. Disjunction is not so easy. If

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdots & \vdots \\ \Gamma, A \vee B @ \mathrm{w}', A @ \mathrm{w}' \Longrightarrow C @ \mathrm{w} & \Gamma, A \vee B @ \mathrm{w}', B @ \mathrm{w}' \Longrightarrow C @ \mathrm{w}\end{array}}{\Gamma, A \vee B @ \mathrm{w}' \Longrightarrow C @ \mathrm{w}} \vee \mathrm{L}$$

we have by induction hypothesis

$$\begin{array}{rcl} \mathcal{E}_1 & :: & \Gamma', x{:}A \vee B @ \mathrm{w}', y{:}A @ \mathrm{w}' \vdash M_1 : C @ \mathrm{w} \\ \mathcal{E}_2 & :: & \Gamma', x{:}A \vee B @ \mathrm{w}', y{:}B @ \mathrm{w}' \vdash M_2 : C @ \mathrm{w} \end{array}$$

and would like to apply $\vee$ E as in

$$\begin{array}{rcl} & & \mathtt{case}\ x\ \mathtt{of} \\ N & = & \quad \mathtt{inl}\ y \Rightarrow M_1 \\ & & \quad \mathtt{inr}\ y \Rightarrow M_2 \end{array}$$

but cannot because the case object x must be at the same world as the conclusions M_1 and M_2 in our local-only rule. We cannot use get either; $A \vee B$ is only mobile when A and B are mobile. Fortunately, the IS5$^\cup$ rule that allows "action at a distance" is in fact derivable in Lambda 5. The proof we use is

```
case (get[w'] case x of
        inl z ⇒ inl (hold z)
        inr z ⇒ inr (hold z)) of
    inl y' ⇒ leta y = y'
             in M1
             end
    inr y' ⇒ leta y = y'
             in M2
             end
```

What we do is perform a remote case analysis at w' in order to turn the hypothesis $x{:}A \vee B @ \mathrm{w}'$ into a proof of $(A\ \mathtt{at}\ \mathrm{w}') \vee (B\ \mathtt{at}\ \mathrm{w}') @ \mathrm{w}'$. We can then get that proof because it is now guaranteed to be mobile. At w we perform another case analysis, unwrap the `at` modality to bind remote hypotheses, and continue with M_1 or M_2. Note that this proof is somewhat odd, in that we enlist the `at` connective, which does not necessarily appear elsewhere in the formula. The consequence is that the logical completeness of $\vee$ in Lambda 5 relies on the presence of `at` (at least for this specific proof). This means that the two are not truly orthogonal in the language.

The full Twelf proof of completeness appears in Appendix A.1 as the relations `seqnd` and `hypnd`. □

3.2.4 Equivalence to IS5$^\cup$

To complete the proof that Lambda 5 is equivalent to IS5$^\cup$, we now prove the soundness and completeness of IS5$^\cup$ relative to the sequent calculus (Figure 3.8). These proofs are easier than the ones from the previous section.

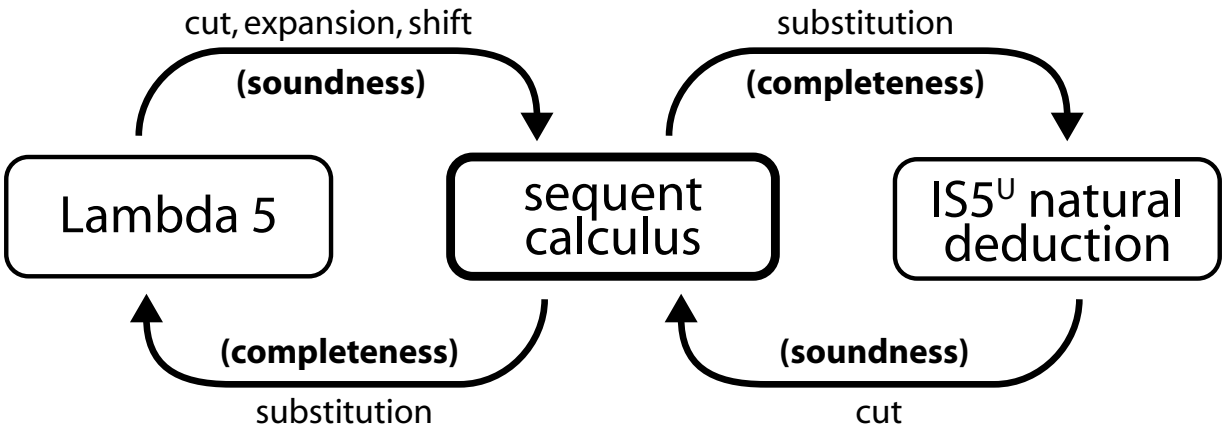

Figure 3.8: The soundness and completeness argument for Lambda 5. Lambda 5 is equivalent to the IS5$^\cup$ sequent calculus (Theorems 5 and 6), which is itself equivalent to IS5$^\cup$ (Theorems 9 and 10).

Theorem 9 (Soundness of IS5$^\cup$ relative to sequent calculus)
If $\mathcal{D} :: \Gamma \vdash A@\mathrm{w}$
then $\mathcal{E} :: \Gamma \Longrightarrow A@\mathrm{w}$

Theorem 10 (Completeness of IS5$^\cup$ relative to sequent calculus)
If $\mathcal{D} :: \Gamma \Longrightarrow A@\mathrm{w}$
then $\mathcal{E} :: \Gamma \vdash A@\mathrm{w}$

Proof of Theorem 9 is the same as Theorem 5, but we do not require shift or expansion since IS5$^\cup$ does not have get. The proof of Theorem 10 follows the proof of Theorem 6, but is easier because the rules are not restricted to be local. Both proofs appear in Appendix A.2 as the relations `nndseq` and `sseqnd`. □

Theorem 11 (Equivalence of Lambda 5 and IS5$^\cup$)

	$\Gamma \vdash M : A@\mathrm{w}$	(in Lambda 5, for some M)
if and only if	$\Gamma \vdash A@\mathrm{w}$	(in IS5$^\cup$)

Equivalence is an easy corollary of Theorems 5, 6, 9, and 10. □

We spent a lot of effort proving that Lambda 5 natural deduction is equivalent to the more straightforward formulation IS5$^\cup$. The purpose of this was to create a logic whose proof terms were a lambda calculus with a simple dynamic semantics that can be used to write distributed applications. In the next section I describe this semantics.

3.3 Dynamic semantics

I give Lambda 5's dynamic semantics here in terms of a big-step evaluation relation $\Downarrow$. Other ways of presenting the dynamic semantics are possible; in Section 3.4.6 I will give a dynamic semantics in terms of continuations that lets us be more explicit about

$$\text{values} \quad v \quad ::= \quad \langle v_1, v_2 \rangle \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v \mid \boldsymbol{\lambda} x.M \mid \mathbf{box}\ \omega.M \mid \mathbf{there}[\mathrm{w}, v] \mid \mathbf{held}\ v \mid x$$

$$\frac{\Gamma \vdash v_1 : A@\mathrm{w} \quad \Gamma \vdash v_2 : B@\mathrm{w}}{\Gamma \vdash \langle v_1, v_2 \rangle : A \wedge B@\mathrm{w}} \wedge \mathrm{V}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathbf{inl}\ v : A \vee B@\mathrm{w}} \vee \mathrm{V}_1 \qquad \frac{\Gamma \vdash v : B@\mathrm{w}}{\Gamma \vdash \mathbf{inr}\ v : A \vee B@\mathrm{w}} \vee \mathrm{V}_2$$

$$\frac{\Gamma, x{:}A@\mathrm{w} \vdash M : B@\mathrm{w}}{\Gamma \vdash \boldsymbol{\lambda} x.M : A \supset B@\mathrm{w}} \supset \mathrm{V} \qquad \frac{\Gamma \vdash v : A@\mathrm{w}'}{\Gamma \vdash \mathbf{held}\ v : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{V}$$

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash M : A@\omega}{\Gamma \vdash \mathbf{box}\ \omega.M : \Box A@\mathrm{w}} \Box \mathrm{V} \qquad \frac{\Gamma \vdash v : A@\mathbf{w}'}{\Gamma \vdash \mathbf{there}[\mathbf{w}', v] : \Diamond A@\mathrm{w}} \Diamond \mathrm{V}$$

$$\frac{}{\Gamma, x{:}A@\mathrm{w} \vdash x : A@\mathrm{w}}\ \mathrm{hyp} \qquad \frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathtt{val}\ v : A@\mathrm{w}}\ \mathrm{val}$$

Figure 3.9: The syntax for Lambda 5 values and their typing rules.

the transfer of control between worlds, as well as give meaning to non-terminating programs. In our previous paper on Lambda 5 we give a continuation-based semantics that explicitly stores references to remote objects in tables [89], similar to the way we implement marshaling (Section 5.5.4). For this discussion our main concern is defining the location in which computation takes place. The $\Downarrow$ relation suffices for that purpose and is briefer to define than a small-step relation.

The evaluation relation $\Downarrow$ is between an expression and a value, and is indexed by the (concrete) world in which the evaluation takes place:

$$M \Downarrow_{\mathbf{w}} v$$

Expressions are the proof terms from Lambda 5, and their typing rules are the inference rules in Figure 3.5. We provide a different syntactic class for values v, and define typing for them via a judgment $\Gamma \vdash v : A@\mathrm{w}$ for them. These appear in Figure 3.9.

Variables are now considered values, which changes the meaning of the expression typing rules (the context now contains hypotheses about the typing of values, not expressions), but we do not repeat the rules here because they would be syntactically identical. The val rule allows us to use a value as an expression of the same type. The main thing to notice about the value typing rules is that they are non-local, like the introduction rules from IS5$^{\cup}$. (For example, at V allows the value within it to be typed at

$$\frac{M \Downarrow_{\mathbf{w}} \langle v_1, v_2 \rangle}{\#1\ M \Downarrow_{\mathbf{w}} v_1}\ \#1\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \langle v_1, v_2 \rangle}{\#2\ M \Downarrow_{\mathbf{w}} v_2}\ \#2\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} v_1 \quad N \Downarrow_{\mathbf{w}} v_2}{\langle M, N \rangle \Downarrow_{\mathbf{w}} \langle v_1, v_2 \rangle}\ \langle , \rangle \Downarrow$$

$$\frac{M \Downarrow_{\mathbf{w}} v}{\mathtt{inl}\ M \Downarrow_{\mathbf{w}} \mathbf{inl}\ v}\ \text{inl}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} v}{\mathtt{inr}\ M \Downarrow_{\mathbf{w}} \mathbf{inr}\ v}\ \text{inr}\ \Downarrow$$

$$\frac{M \Downarrow_{\mathbf{w}} \mathbf{inl}\ v_1 \quad [v_1/x]N_1 \Downarrow_{\mathbf{w}} v}{\mathtt{case}\ M\ \mathtt{of}\ \mathtt{inl}\ x \Rightarrow N_1 \mid \mathtt{inr}\ x \Rightarrow N_2 \Downarrow_{\mathbf{w}} v}\ \text{case}\ \Downarrow_1$$

$$\frac{M \Downarrow_{\mathbf{w}} \mathbf{inr}\ v_2 \quad [v_2/x]N_2 \Downarrow_{\mathbf{w}} v}{\mathtt{case}\ M\ \mathtt{of}\ \mathtt{inl}\ x \Rightarrow N_1 \mid \mathtt{inr}\ x \Rightarrow N_2 \Downarrow_{\mathbf{w}} v}\ \text{case}\ \Downarrow_2$$

$$\frac{}{\lambda x.M \Downarrow_{\mathbf{w}} \boldsymbol{\lambda} x.M}\ \lambda\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \boldsymbol{\lambda} x.M' \quad N \Downarrow_{\mathbf{w}} v' \quad [v'/x]M' \Downarrow_{\mathbf{w}} v}{M\ N \Downarrow_{\mathbf{w}} v}\ \text{app}\ \Downarrow$$

$$\frac{}{\mathtt{box}\ \omega.M \Downarrow_{\mathbf{w}} \mathbf{box}\ \omega.M}\ \text{box}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \mathbf{box}\ \omega.N \quad [\mathbf{w}/\omega]N \Downarrow_{\mathbf{w}} v}{\mathtt{unbox}\ M \Downarrow_{\mathbf{w}} v}\ \text{unbox}\ \Downarrow$$

$$\frac{M \Downarrow_{\mathbf{w}} v}{\mathtt{hold}\ M \Downarrow_{\mathbf{w}} \mathbf{held}\ v}\ \text{hold}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \mathbf{held}\ v' \quad [v'/x]N \Downarrow_{\mathbf{w}} v}{\mathtt{leta}\ x = M\ \mathtt{in}\ N \Downarrow_{\mathbf{w}} v}\ \text{leta}\ \Downarrow$$

$$\frac{M \Downarrow_{\mathbf{w}} v}{\mathtt{here}\ M \Downarrow_{\mathbf{w}} \mathbf{there}[\mathbf{w}, v]}\ \text{here}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \mathbf{there}[\mathbf{w}', v'] \quad [\mathbf{w}'/\omega][v'/x]N \Downarrow_{\mathbf{w}} v}{\mathtt{letd}\ \omega, x = M\ \mathtt{in}\ N \Downarrow_{\mathbf{w}} v}\ \text{letd}\ \Downarrow$$

$$\frac{}{\mathtt{val}\ v \Downarrow_{\mathbf{w}} v}\ \text{val}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}'} v}{\mathtt{get}[\mathbf{w}']\ M \Downarrow_{\mathbf{w}} v}\ \text{get}\ \Downarrow$$

Figure 3.10: Lambda 5 big-step dynamic semantics, given as a relation $M \Downarrow_{\mathbf{w}} v$ between an expression M and its value v.

another world.) This is allowable because, as values, they do not need any more evaluation, and therefore require no "action at a distance." Because typing is non-local, we will be able to move certain values from world to world and retain their well-formedness (Theorem 13). Most of the values resemble the expressions that construct them, but a value of type $\Diamond A$ is different. It is written $\mathbf{there}[\mathbf{w}, v]$ where $\mathbf{w}$ is a world constant (the world where A is true) and v is the value well-typed at $\mathbf{w}$. We need both because the elimination form binds a world variable and value variable.

The definition of evaluation appears in Figure 3.10. Evaluation takes place on closed terms so there is no case for evaluating variables (they are replaced with values by substitution). Observe that evaluation only changes worlds in the get $\Downarrow$ rule; all other action happens locally. Additionally, we are always evaluating at a constant world (not a variable), meaning that there is always a specific concrete world at which to carry out computation.

The main theorem about the evaluation relation is that it preserves well-typedness:

Theorem 12 (Type preservation of $\Downarrow$)
If $\mathcal{D} :: \cdot \vdash M : A@\mathbf{w}$
and $\mathcal{E} :: M \Downarrow_{\mathbf{w}} v$
then $\mathcal{F} :: \cdot \vdash v : A@\mathbf{w}$

The proof is by induction on the derivation $\mathcal{E}$, and is completely standard except for the get case. If

$$\mathcal{E} \quad = \quad \frac{M \Downarrow_{\mathbf{w}'} v}{\texttt{get}[\mathbf{w}']\ M \Downarrow_{\mathbf{w}} v}\ \text{get}\ \Downarrow$$

then by inversion on $\mathcal{D}$ we have

$$\mathcal{D}_1 :: A\ \mathsf{mobile} \qquad\qquad \mathcal{D}_2 :: \cdot \vdash M : A@\mathbf{w}'$$

so by induction hypothesis we have

$$\mathcal{E}' :: \cdot \vdash v : A@\mathbf{w}'$$

but require v to be well-typed at the source world $\mathbf{w}$. To prove this we need a lemma that says that a value of mobile type at one world also has that type at other worlds (Theorem 13 below). Applying this lemma, we get

$$\mathcal{F} :: \cdot \vdash v : A@\mathbf{w}$$

as required. The full proof appears implicitly as the well-typedness of the `eval` relation in Appendix A.3. □

The value shift lemma is stated as follows:

Theorem 13 (Value shift)
If $\mathcal{D} :: \cdot \vdash v : A@\mathbf{w}$
and $\mathcal{E} :: A\ \mathsf{mobile}$
then $\mathcal{F} :: \cdot \vdash v : A@\mathbf{w}'$

We consider this the computational justification for the mobile judgment. This proof is a straightforward structural induction over the derivation $\mathcal{E}$ that the type is mobile. There are a few differences between this and the shift theorem (Theorem 4) we used in Section 3.2.3. First, we insist that the *same value* be well-typed at both worlds—in the shift theorem we allowed the proof to be transformed. This allows the dynamic semantics to simply send the value from one world to another without modifying it. Second, we require the value to be closed, so that we do not have to consider the case of variables. The full proof appears in Appendix A.3. □

We also desire a progress theorem for $\Downarrow$ (otherwise we may have "forgotten" some rules), but this is difficult to state for a big-step semantics (because of the possibility that we may make progress but never terminate, and therefore have no finite derivation). In the Twelf formalization we use its coverage checker [121] to prove that some rule of the `eval` relation always immediately applies, even if the proof search process does not terminate. □

The dynamic semantics for Lambda 5 is close to what we want for our programming language. When evaluating locally, the semantics match up with the familiar ones from lambda calculus, giving a clear path to implementation. However, the concepts that it offers are not enough to write and compile real programs. The two features that we wish to add are universal reasoning and continuations. Universal reasoning allows us to avoid explicitly moving values around with get when those values make sense anywhere, so that we are not burdened by the modal type system when we are not using it. The logical basis for this is discussed in Section 3.5. Continuations allow us to compile our language via Continuation Passing Style, which is the basis of our implementation of threads, part of the interface to databases and the GUI, and a useful high-level programming construct. The logical basis for continuations is classical modal logic, which is discussed in the next section.

3.4 C5

Lambda 5 is a computational modal lambda calculus based on intuitionistic S5. In this section I present C5, a calculus based on *classical* S5. This gives us a logical explanation of continuations. Although ML5 supports first-class continuations, ultimately we do not adopt the specific formalism presented in this section, because the full power of network-wide continuations suggested by the logic is too heavyweight for our needs. Therefore, this section can be seen as an excursion, and not necessary for understanding the remainder of the dissertation.

3.4.1 Classical control flow

The notion that control operators such as Scheme's `call/cc` or Felleisen's $\mathcal{C}$ can be given logical meaning via classical logic is well known. Essentially, if we interpret the proposition $\neg A$ as the type of a continuation expecting a value of type A, then the types of these operators are classical tautologies. For example, in Standard ML, the types of the control operators are

$$\begin{array}{lcl} \texttt{callcc} & : & (\alpha\ \texttt{cont} \to \alpha) \to \alpha \\ \texttt{throw} & : & \alpha\ \texttt{cont} \to \alpha \to \beta \end{array}$$

The type of `callcc` interpreted as a proposition is $(\neg A \supset A) \supset A$, a classical theorem similar to Peirce's law. The type of `throw` as a proposition is $\neg A \supset A \supset B$, which is equivalent to $(\neg A \wedge A) \supset \bot$, the law of non-contradiction. Griffin first proposed this in 1990 [47] with later refinements by (for example) Murthy [92]. Parigot's $\lambda\mu$-calculus [99] takes this idea and develops it into a full-fledged natural deduction system for classical logic. C5 is a similar account, extended to the modal case and presented in terms of true and false judgments.

$$\frac{\Gamma \vdash M : A@\mathrm{w} \quad \Gamma \vdash N : B@\mathrm{w}}{\Gamma \vdash \langle M, N\rangle : A \wedge B@\mathrm{w}}\ \wedge \mathrm{I} \qquad \frac{\Gamma \vdash M : A \wedge B@\mathrm{w}}{\Gamma \vdash \#1\, M : A@\mathrm{w}}\ \wedge \mathrm{E}_1 \qquad \frac{\Gamma \vdash M : A \wedge B@\mathrm{w}}{\Gamma \vdash \#2\, M : B@\mathrm{w}}\ \wedge \mathrm{E}_2$$

$$\frac{\Gamma, x{:}A@\mathrm{w} \vdash M : B@\mathrm{w}}{\Gamma \vdash \lambda x.M : A \supset B@\mathrm{w}}\ \supset \mathrm{I} \qquad \frac{\Gamma \vdash M : A \supset B@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}}{\Gamma \vdash M\ N : B@\mathrm{w}}\ \supset \mathrm{E}$$

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{hold}\, M : A\,\mathtt{at}\,\mathrm{w}@\mathrm{w}}\ \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma \vdash M : A\,\mathtt{at}\,\mathrm{w}'@\mathrm{w} \quad \Gamma, x{:}A@\mathrm{w}' \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{leta}\, x = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \mathtt{at}\ \mathrm{E}$$

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash M : A@\omega}{\Gamma \vdash \mathtt{box}\, \omega.M : \Box A@\mathrm{w}}\ \Box \mathrm{I} \qquad \frac{\Gamma \vdash M : \Box A@\mathrm{w}}{\Gamma \vdash \mathtt{unbox}\, M : A@\mathrm{w}}\ \Box \mathrm{E}$$

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{here}\, M : \Diamond A@\mathrm{w}}\ \Diamond \mathrm{I} \qquad \frac{\Gamma \vdash M : \Diamond A@\mathrm{w} \quad \Gamma, \omega\ \mathsf{world}, x{:}A@\omega \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{letd}\, \omega, x = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \Diamond \mathrm{E}$$

$$\frac{}{\Gamma, x{:}A@\mathrm{w} \vdash x : A@\mathrm{w}}\ \mathrm{hyp} \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : A@\mathrm{w}'}{\Gamma \vdash \mathtt{get}[\mathrm{w}']\, M : A@\mathrm{w}}\ \mathrm{get}$$

$$\frac{\Gamma \vdash M : \bot@\mathrm{w}}{\Gamma \vdash \mathtt{abort}\, M : C@\mathrm{w}}\ \bot \mathrm{E}$$

$$\frac{\Gamma, u{:}A\star\mathrm{w} \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{letcc}\, u\ \mathtt{in}\ M : A@\mathrm{w}}\ \mathrm{bc} \qquad \frac{\Gamma, u{:}A\star\mathrm{w} \vdash M : A@\mathrm{w}}{\Gamma, u{:}A\star\mathrm{w} \vdash \mathtt{throw}\, M\ \mathtt{to}\ u : C@\mathrm{w}'}\ \#$$

Figure 3.11: C5 natural deduction, based on classical S5.

3.4.2 Classical natural deduction

Classical logic exhibits a duality between truth (proofs) and falsehood (refutations). We embody this duality by defining two judgments on propositions:

$$A\ \mathsf{true}@\mathrm{w} \qquad\qquad A\ \mathsf{false}\star\mathrm{w}$$

The first (abbreviated as $A@\mathrm{w}$) says that the proposition A is true at the world w; a witness to this fact is a proof of A, or an expression in our programming language. The second, abbreviated $A\star\mathrm{w}$, says that A is false at the world w; a witness to this fact is a refutation of A, or a continuation in our programming language.

We could contemplate giving a complete description of the truth and falsehood judgments independently. If we did so, our logic would have some redundancy because of the duality of truth and falsehood: for any proof of $A@\mathrm{w}$, we could equally well refute $\neg A \star \mathrm{w}$. Instead, we want to isolate a particular form of reasoning into either the truth or falsehood judgment, leading to a more parsimonious logic. Some systems work by allocating rules equally to the two judgments; for example in Wadler's dual calculus [140, 141] and the sequent calculus in Section 3.4.3 the left rules all operate on truth, and the right rules all operate on falsehood. Nanevski gives a classical natural

deduction system in which there are no elimination rules, only introduction of falsehood and truth [94]. In each case, the expressiveness of the system comes from the structural properties that allow for the mixing of proofs and refutations. In C5 natural deduction we take a deliberately asymmetric view: We push *all* of the reasoning into the truth judgment and provide only the structural rules for interfacing with the degenerate falsehood judgment. This is because programming with truth is programming with values—a more familiar style than manipulating continuations. The calculus that results is just Lambda 5 extended with two structural rules, which are essentially `callcc` and `throw`. We prove that this formulation of the logic sacrifices no expressiveness.

$$\frac{}{\Box A \text{ mobile}}\ \Box\,\mathrm{M} \qquad \frac{}{\Diamond A \text{ mobile}}\ \Diamond\,\mathrm{M}$$

$$\frac{}{A\ \mathtt{at}\ \mathrm{w} \text{ mobile}}\ \mathtt{at}\ \mathrm{M}$$

$$\frac{A \text{ mobile} \quad B \text{ mobile}}{A \wedge B \text{ mobile}}\ \wedge\,\mathrm{M} \qquad \frac{}{\bot \text{ mobile}}\ \bot\,\mathrm{M}$$

Figure 3.12: Definition of the mobile judgment for C5.

C5 natural deduction appears in Figure 3.11. The rules for $\wedge$, $\supset$, $\Box$, `at`, and $\Diamond$ are identical to those of Lambda 5. We omit $\vee$, since it can be defined in terms of $\supset$, $\bot$, and $\wedge$, and anyway would be treated just as in Lambda 5. The only new connective is $\bot$. It has no introduction form, and its elimination form (Rule $\bot$ E) is restricted to be local as usual. The $\bot$ proposition is mobile, however (Figure 3.12).

In C5 the context Γ can contain hypotheses of the form ω world, $x{:}A@\mathrm{w}$, and $u{:}A\star\mathrm{w}$. Falsehood hypotheses are introduced by the rule bc ("by contradiction"), which corresponds directly to the classical axiom $(\neg A \supset A) \supset A$. Operationally, `letcc` captures the current continuation (which expects a value of type $A@\mathrm{w}$) and binds it as a continuation variable $A\star\mathrm{w}$ while continuing the proof of $A@\mathrm{w}$. The $\#$ rule may be alarming at first glance because it requires the assumption $A\star\mathrm{w}$ to appear in the conclusion. This is because the $\#$ rule is actually the *hypothesis rule* for falsehood, and will have a corresponding substitution principle. The rule simply states that if we have the assumption that A is false and are able to prove that A is true at the same world, then we can deduce a contradiction and thus any proposition. The $\#$ rule is realized operationally as a throw of an expression to a matching continuation. Note that continuations are *global*—we can throw from any world to a remote continuation $A\star\mathrm{w}$, provided that we are able to construct a proof of $A@\mathrm{w}$.

For each kind of hypothesis we have a substitution theorem.

Theorem 14 (C5 truth substitution)
If $\mathcal{D} :: \Gamma \vdash M : A@\mathrm{w}$
and $\mathcal{E} :: \Gamma, x{:}A@\mathrm{w} \vdash N : B@\mathrm{w}'$
then $\mathcal{F} :: \Gamma \vdash [{}^{M}/_{x}]N : B@\mathrm{w}'$

Theorem 15 (C5 falsehood substitution)
If $\forall C, \omega.\ \mathcal{D} :: \Gamma, x{:}A@\mathrm{w} \vdash M : C@\omega$
and $\mathcal{E} :: \Gamma, u{:}A\star\mathrm{w} \vdash N : B@\mathrm{w}'$
then $\mathcal{F} :: \Gamma \vdash [\![{}^{x.M}/_{u}]\!] N : B@\mathrm{w}'$

Here, truth substitution $[{}^{M}/_{x}]N$ is defined as in Lambda 5. Theorem 15, however, warrants special attention. This principle is dual to the # rule just as Theorem 14 is dual to hyp. The # rule contradicts an $A\star\mathrm{w}$ with an $A@\mathrm{w}$, so to eliminate a falsehood assumption by substitution we are able to assume $A@\omega$ and must produce another contradiction. Reading $\vdash$ as logical consequence, we have that if A false gives B, and A true gives C (for all C), then B. This can easily be seen as a consequence of excluded middle on A. We write this substitution as $[\![{}^{x.M}/_{u}]\!] N$ where x is a binder (with scope through M) that stands for the value thrown to u. It is defined pointwise on N except for a use of the # rule on u:

$$[\![{}^{x.M}/_{u}]\!] \texttt{throw}\ N'\ \texttt{to}\ u \ \doteq\ [{}^{N'}/_{x}]M$$

This principle is close to what Parigot calls *structural substitution* for the $\lambda\mu$-calculus. Operationally, we see this as replacing the throw with some other handler for A. Since the new handler must have parametric type, typically it is a `throw` to some other continuation, perhaps after performing some computation on the proof of A.

Proof of Theorem 14 is by a trivial induction on $\mathcal{E}$ (it comes "for free" in the Twelf formalization in Appendix A.4). Proof of Theorem 15 is by induction on the derivation $\mathcal{E}$, appealing to Theorem 14 in the case above. The full proof is given as the relation `xs` in Appendix A.4. □

We wish to know that our proof theory (specially constructed to give rise to a good operational semantics) is not simply ad hoc; that it is consistent and really embodies classical S5. To do so we prove in the next section a correspondence to a straightforward sequent formulation of classical S5 with the subformula property. We'll use the sequent calculus as intuition as we develop proof terms for some classically true propositions in Section 3.4.5, and then explore the dynamic semantics of the language in Section 3.4.6.

3.4.3 Classical sequent calculus

The classical S5 sequent calculus appears in Figure 3.13. This logic is motivated by symmetry and simplicity alone, so we do not include proof terms, restrict rules to act locally, or bias the reasoning towards truth or falsehood. Because we want a system without bias, the judgment should have a symmetric reading. Therefore we do not prove truth from falsehood assumptions, nor refute falsehoods from truth assumptions; instead, hypotheses of truth and falsehood come together to derive mutual contradiction. We write this judgment as

$$\Gamma \ \#\ \Delta$$

where Γ contains hypotheses of the truth of propositions, and Δ contains hypotheses of the falsehood of propositions [106]. The contra rule allows us to derive a contradiction from a matching true and false hypothesis. The remainder of the rules show us how to

$$\frac{}{\Gamma, A@\mathrm{w} \mathrel{\#} A\star\mathrm{w}, \Delta}\ \text{contra} \qquad \frac{}{\Gamma, \bot@\mathrm{w} \mathrel{\#} \Delta}\ \bot\ \mathrm{T}$$

$$\frac{\Gamma, A \supset B@\mathrm{w}, B@\mathrm{w} \mathrel{\#} \Delta \quad \Gamma, A \supset B@\mathrm{w} \mathrel{\#} A\star\mathrm{w}, \Delta}{\Gamma, A \supset B@\mathrm{w} \mathrel{\#} \Delta}\ \supset\ \mathrm{T} \qquad \frac{\Gamma, A@\mathrm{w} \mathrel{\#} B\star\mathrm{w}, A \supset B\star\mathrm{w}, \Delta}{\Gamma \mathrel{\#} A \supset B\star\mathrm{w}, \Delta}\ \supset\ \mathrm{F}$$

$$\frac{\Gamma, \Box A@\mathrm{w}, A@\mathrm{w}' \mathrel{\#} \Delta}{\Gamma, \Box A@\mathrm{w} \mathrel{\#} \Delta}\ \Box\ \mathrm{T} \qquad \frac{\Gamma, \omega\ \mathsf{world} \mathrel{\#} A\star\omega, \Box A\star\mathrm{w}, \Delta}{\Gamma \mathrel{\#} \Box A\star\mathrm{w}, \Delta}\ \Box\ \mathrm{F}$$

$$\frac{\Gamma, \Diamond A@\mathrm{w}, \omega\ \mathsf{world}, A@\omega \mathrel{\#} \Delta}{\Gamma, \Diamond A@\mathrm{w} \mathrel{\#} \Delta}\ \Diamond\ \mathrm{T} \qquad \frac{\Gamma \mathrel{\#} A\star\mathrm{w}', \Diamond A\star\mathrm{w}, \Delta}{\Gamma \mathrel{\#} \Diamond A\star\mathrm{w}, \Delta}\ \Diamond\ \mathrm{F}$$

$$\frac{\Gamma, A \wedge B@\mathrm{w}, A@\mathrm{w}, B@\mathrm{w} \mathrel{\#} \Delta}{\Gamma, A \wedge B@\mathrm{w} \mathrel{\#} \Delta}\ \wedge\ \mathrm{T} \qquad \frac{\Gamma \mathrel{\#} A\star\mathrm{w}, A \wedge B\star\mathrm{w}, \Delta \quad \Gamma \mathrel{\#} B\star\mathrm{w}, A \wedge B\star\mathrm{w}, \Delta}{\Gamma \mathrel{\#} A \wedge B\star\mathrm{w}, \Delta}\ \wedge\ \mathrm{F}$$

$$\frac{\Gamma, A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}, A@\mathrm{w}' \mathrel{\#} \Delta}{\Gamma, A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w} \mathrel{\#} \Delta}\ \mathtt{at}\ \mathrm{T} \qquad \frac{\Gamma \mathrel{\#} A\star\mathrm{w}', A\ \mathtt{at}\ \mathrm{w}'\star\mathrm{w}, \Delta}{\Gamma \mathrel{\#} A\ \mathtt{at}\ \mathrm{w}'\star\mathrm{w}, \Delta}\ \mathtt{at}\ \mathrm{F}$$

Figure 3.13: Classical S5 sequent calculus. Left rules operate exclusively on truth hypotheses in Γ, and right rules operate exclusively on falsehood hypotheses in Δ. The sequent $\Gamma \mathrel{\#} \Delta$ is read as follows: given hypotheses of truth Γ and hypotheses of falsehood Δ, we derive a contradiction. We include hypotheses about the existence of worlds in Γ, as a convention.

decompose a connective in either the truth or falsehood context. It is best to read these rules bottom-up, as if during proof search. For example, in the $\bot$ T rule, $\bot$ being true at any world is contradictory, so we are done. In $\wedge$ T, to reason from the hypothesis that $A \wedge B$ is true, we assume that A is true and B is true. In $\wedge$ F, $A \wedge B$ being false would mean that either A is false or B is false, so we must show contradictions in each of those cases. Note that $\Box$ and $\Diamond$ are dual in this presentation, and that the `at` connective is self-dual.

Because each rule analyzes exactly one connective by breaking it into its components, this sequent calculus also has the subformula property. This gives us consistency:

Theorem 16 (Consistency of classical S5 sequent calculus)
Not all sequents are provable.

Proof is immediate, by counterexample. Consider the sequent $\cdot \mathrel{\#} \bot\star\mathbf{w}$ for some world constant $\mathbf{w}$. There is no rule for decomposing $\bot$ in the falsehood context, so there is no proof. □

3.4.4 Soundness and completeness

We would now like to prove that our local, truth-biased natural deduction system is equivalent to the sequent calculus. We start by proving soundness. Like the soundness of Lambda 5 with respect to the IS5$^{\cup}$ sequent calculus (Theorem 11) we require a cut lemma about the sequent calculus. For the classical sequent calculus presented in this symmetric manner, this lemma is *excluded middle*:

Theorem 17 (Excluded middle)

If	$\mathcal{D}$::	$\Gamma, A@\mathrm{w} \# \Delta$
and	$\mathcal{E}$::	$\Gamma \# A\star\mathrm{w}, \Delta$
then	$\mathcal{F}$::	$\Gamma \# \Delta$

Proof of Theorem 17 is by lexicographic induction over the structure of the formula A and (simultaneously) the derivations $\mathcal{D}$ and $\mathcal{E}$. The interesting cases are when $\mathcal{D}$ ends with a truth rule acting on A and $\mathcal{E}$ is a falsehood rule acting on A. For example, if

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma, \Box A@\mathrm{w}, A@\mathrm{w}' \# \Delta\end{array}}{\Gamma, \Box A@\mathrm{w} \# \Delta}\,\Box\,\mathrm{T} \qquad \text{and } \mathcal{E} \quad = \quad \dfrac{\begin{array}{c}\mathcal{E}' \\ \vdots \\ \Gamma, \omega\ \mathsf{world} \# A\star\omega, \Box A\star\mathrm{w}, \Delta\end{array}}{\Gamma \# \Box A\star\mathrm{w}, \Delta}\,\Box\,\mathrm{F}$$

then by induction hypothesis (on $\Box A$, weaken($\mathcal{D}$), and weaken($\mathcal{E}'$)) we have

$$\mathcal{E}'' :: \Gamma, \omega\ \mathsf{world} \# A\star\omega, \Delta$$

and by induction hypothesis (on $\Box A$, $\mathcal{D}'$, and weaken($\mathcal{E}$)) we have

$$\mathcal{D}'' :: \Gamma, A@\mathrm{w}' \# \Delta$$

so by world substitution we get

$$[{}^{\mathrm{w}'}/_{\omega}]\mathcal{E}'' :: \Gamma \# A\star\mathrm{w}', \Delta$$

and then by induction hypothesis (on A, $\mathcal{D}''$, $[{}^{\mathrm{w}'}/_{\omega}]\mathcal{E}''$) we have

$$\mathcal{F}'' :: \Gamma \# \Delta$$

as required.

The full proof appears in Appendix A.4 as the relation `xm`. □

We also require a lemma that justifies the mobile judgment. Its statement is

Theorem 18 (Switch)

If	$\mathcal{D}$	::	A mobile
and	$\mathcal{E}$	::	$\Gamma \# A\star\mathrm{w}, \Delta$
then	$\mathcal{F}$	::	$\Gamma \# A\star\mathrm{w}', \Delta$

The proof is similar to Theorem 4, proceeding by induction on $\mathcal{D}$ and appealing to excluded middle. It appears as the relation `switch` in Appendix A.4. □

We can now state and prove soundness of C5:

Theorem 19 (C5 soundness)
If $\mathcal{D} :: \Gamma \vdash M : A@\mathrm{w}$
then $\mathcal{F} :: \Gamma^{@} \# A\star\mathrm{w}, \Delta^{\star}$

$\Gamma^{@}$ selects all of the hypotheses of the form $B@\mathrm{w}'$ (and ω world) from the context, and $\Gamma^{\star}$ selects the hypotheses of the form $B\star\mathrm{w}'$. Soundness states that if from some hypotheses we can derive a conclusion of A being true at w, then in the sequent calculus we can derive a contradiction from those same hypotheses, plus a hypothesis that A is false at w. (Operationally, we can think of falsehood assumption as the "final continuation" that takes the result of our computation.)

The proof of soundness is by induction on $\mathcal{D}$. The elimination rules are straightforward because they match the F rules in the sequent calculus. The get rule appeals to Theorem 18. Elimination rules appeal to excluded middle. Interestingly, the structural rules bc and # simply use contraction and weakening for the sequent calculus. If

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma, u{:}A\star\mathrm{w} \vdash M : A@\mathrm{w}\end{array}}{\Gamma \vdash \mathtt{letcc}\ u\ \mathtt{in}\ M : A@\mathrm{w}}\ \mathrm{bc}$$

then by induction hypothesis we have

$$\mathcal{D}' :: \Gamma^{@} \# A\star\mathrm{w}, A\star\mathrm{w}, \Gamma^{\star}$$

which gives us $\Gamma^{@} \# A\star\mathrm{w}, \Gamma^{\star}$ by contraction as required. If

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma, u{:}A\star\mathrm{w} \vdash M : A@\mathrm{w}\end{array}}{\Gamma, u{:}A\star\mathrm{w} \vdash \mathtt{throw}\ M\ \mathtt{to}\ u : C@\mathrm{w}'}\ \#$$

then we have by induction hypothesis

$$\mathcal{D}' :: \Gamma^{@} \# A\star\mathrm{w}, A\star\mathrm{w}, \Gamma^{\star}$$

which gives us $\Gamma^{@} \# A\star\mathrm{w}, \Gamma^{\star}$ by contraction and then $\Gamma^{@} \# A\star\mathrm{w}, C\star\mathrm{w}', \Gamma^{\star}$ by weakening as required. The full proof of Theorem 19 appears as the relations `contfalse` and `ndseq` in Appendix A.4. □

The other half of the equivalence argument is completeness:

Theorem 20 (C5 completeness)
If $\mathcal{D} :: \Gamma \# \Delta$
then $\mathcal{F} :: \Gamma, \Delta \vdash M : C@\omega$ for some M, for all C, ω

The theorem statement is that if Γ and Δ are contradictory hypotheses, then we can prove any proposition true at any world in the natural deduction, under those same hypotheses. Proof is by induction on $\mathcal{D}$. Uses of T rules are easy; they correspond directly to the elimination rules in natural deduction. However, since the natural deduction is

biased towards reasoning about truth rather than falsehood, the F rules are more difficult and make nontrivial use of the falsehood substitution theorem (Theorem 15). For instance, in the $\wedge F$ case we have by induction hypothesis:

$$\Gamma, up{:}A \wedge B \star \mathrm{w}, ua{:}A \star \mathrm{w} \vdash N_1 : C \mathbin{@} \omega \quad (\forall C, \omega)$$
$$\Gamma, up{:}A \wedge B \star \mathrm{w}, ub{:}B \star \mathrm{w} \vdash N_2 : C \mathbin{@} \omega \quad (\forall C, \omega)$$

By two applications of Theorem 15, we get that the following proof term has any type at any world:

$$\left[\!\!\left[{}^{x.[\![{}^{y.\mathtt{throw}\langle x,y\rangle \ \mathtt{to}\ up}/_{ub}]\!] N_2}/_{ua} \right]\!\!\right] N_1$$

First, we form a `throw` of the pair $\langle x, y\rangle$ to our pair continuation up. This has free truth hypotheses $x{:}A$ and $y{:}B$. Therefore, we can use it to substitute away the ub continuation in N_2 (any throw of M to ub becomes a throw of $\langle x, M\rangle$ to up). Finally, we can use this new term to substitute away ua in N_1, giving us a term that depends only on the pair continuation up. This pattern of *prepending* work onto continuations through substitution is characteristic of this proof, and reflects our bias towards the truth judgment in natural deduction. As another example, in the case for the $\Diamond F$ rule we have by induction hypothesis:

$$\Gamma, u{:}A \star \mathrm{w}', ud{:}\Diamond A \star \mathrm{w} \vdash N : C \mathbin{@} \omega \quad (\forall C, \omega)$$

Our proof term in natural deduction is then:

$$[\![{}^{x.\mathtt{throw}\ (\mathtt{get}[\mathrm{w}'](\mathtt{here}\ x))\ \mathtt{to}\ ud}/_{u}]\!] N$$

Simply enough, if u is ever thrown to, then we instead take that term's address (which lives at w'), move it to w, and throw it to our $\Diamond A$ continuation ud.

Finally, the case for $\Box F$ is interesting because it involves a `letcc`. By induction hypothesis we have:

$$\Gamma, \omega' \ \mathsf{world}, u{:}A \star \omega', ub{:}\Box A \star \mathrm{w} \vdash N : C \mathbin{@} \omega \quad (\forall C, \omega)$$

Then the proof term witnessing the theorem here is:

$$\mathtt{throw}\ (\mathtt{box}\ \omega'.\mathtt{letcc}\ u\ \mathtt{in}\ N)\ \mathtt{to}\ ub$$

It is not possible to use falsehood substitution on u in this case. To do so we would need to turn a term of type $A \mathbin{@} \omega'$ into a $\Box A \mathbin{@} \omega$ to throw to ub. Although at a meta-level we know that we can choose any ω', it won't be possible to internalize this in order to create a proof of $\Box A$. Instead we must introduce a new box, and choose ω' to be the new hypothetical world that the $\Box I$ rule introduces. At that point we use `letcc` to create a real $A \star \omega'$ assumption to discharge u. The full proof appears in Appendix A.4 as the relations `truend` and `falsend`.[2] □

[2]The most natural LF encoding of falsehood is 3rd-order [4]; we use a 2nd-order encoding in our proofs (proving the falsehood substitution theorem by hand) because third-order proof checking is not available in Twelf.

The rule ... is admissible as ...

$$\frac{\Gamma, \neg A @ \mathrm{w} \mathrel{\#} A \star \mathrm{w}, \Delta}{\Gamma, \neg A @ \mathrm{w} \mathrel{\#} \Delta}\ \neg\,\mathrm{T} \qquad \frac{\Gamma, A \supset \bot @ \mathrm{w} \mathrel{\#} A \star \mathrm{w}, \Delta \quad \overline{\Gamma, A \supset \bot @ \mathrm{w}, \bot @ \mathrm{w} \mathrel{\#} \Delta}\ \bot\,\mathrm{T}}{\Gamma, A \supset \bot @ \mathrm{w} \mathrel{\#} \Delta}\ \supset\,\mathrm{T}$$

$$\frac{\Gamma, A @ \mathrm{w} \mathrel{\#} \neg A \star \mathrm{w}, \Delta}{\Gamma \mathrel{\#} \neg A \star \mathrm{w}, \Delta}\ \neg\,\mathrm{F} \qquad \frac{\dfrac{\Gamma, A @ \mathrm{w} \mathrel{\#} A \supset \bot \star \mathrm{w}, \Delta}{\Gamma, A @ \mathrm{w} \mathrel{\#} \bot \star \mathrm{w}, A \supset \bot \star \mathrm{w}, \Delta}\ \text{weaken-}\bot}{\Gamma \mathrel{\#} A \supset \bot \star \mathrm{w}, \Delta}\ \supset\,\mathrm{F}$$

Figure 3.14: The admissible rules for negation in the C5 sequent calculus and natural deduction.

Equivalence. Putting Theorems 19 and 20 together, we have that $\Gamma \vdash M : A @ \mathrm{w}$ gives $\Gamma^{@} \mathrel{\#} A \star \mathrm{w}, \Gamma^{\star}$ (by soundness) which then gives $\forall C, \omega. \Gamma^{@}, \Gamma^{\star}, u{:}A \star \mathrm{w} \vdash M' : C @ \omega$. Observe that $\Gamma^{@}, \Gamma^{\star}$ is Γ, and by choosing $C = A$ and $\omega = \mathrm{w}$ we have $\Gamma, u{:}A \star \mathrm{w} \vdash M' : A @ \mathrm{w}$. Then by application of the rule bc we are back to the original judgment $\Gamma \vdash \texttt{letcc}\ u\ \texttt{in}\ M' : A @ \mathrm{w}$ (with a normalized proof term). Thus $\vdash$ and $\#$ are really equivalent.

3.4.5 Examples

Before presenting the operational semantics for C5, it may be helpful to see some example proof terms and their intended computational meaning. Because our examples use negation ($\neg A$), we'll need to briefly explain how we treat it.

Negation

Although we have not given the rules for the negation connective, it is easily added to the system. Here we equivalently take the standard shortcut of treating $\neg A$ as an abbreviation for $A \supset \bot$. We computationally read $\neg A @ \omega$ as a continuation expecting A, although this should be distinguished from primitive continuations u with type $A \star \omega$: the former is formed by lambda abstraction and eliminated by application, while the latter is formed with `letcc` and eliminated by a `throw` to it. The two are related in that we can reify a continuation assumption $u{:}A \star \omega$ as a negated formula $\neg A$ by lambda abstracting a throw to it: $\lambda a.\texttt{throw}\ a\ \texttt{to}\ u$. Likewise, we can get a falsehood assumption from a term M of type $\neg A$, namely $M(\texttt{letcc}\ u\ \texttt{in}\ \ldots)$.

Finally, note that we have admissible sequent calculus rules $\neg T$ and $\neg F$. Each just flips the proposition under negation to the other side of the sequent, as expected (Figure 3.14).

Classical axioms

Let's begin with the following classical equivalence

$$\Box A \equiv \neg\Diamond\neg A$$

(In fact, in classical logic it is standard practice to *define* $\Box$ this way). From left to right the implication is intuitionistically provable, so we'll look at the proof of the implication from right to left. We begin with the sequent calculus proof, to show why this is clearly true classically. We elide any residual assumptions that go unused.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{-, A@\omega' \# A@\omega', -}\,\text{contra}}{- \# \neg A@\omega', A@\omega', -}\,\neg\,\text{F}}{- \# \Diamond\neg A@\omega, A@\omega', -}\,\Diamond\,\text{F}}{- \# \Diamond\neg A@\omega, \Box A@\omega}\,\Box\,\text{F}}{\neg\Diamond\neg A@\omega \# \Box A@\omega}\,\neg\,\text{T}}{\# \neg\Diamond\neg A \supset \Box A@\omega}\,\supset\,\text{F}$$

In this proof, we are using $\Box$ F to get the hypothetical world ω' at which $\Box A$ is false. From there, we can learn $\neg A$ at the same world, which leads to a contradiction. In natural deduction, the proof tells an interesting story:

$$\begin{array}{ll}
\lambda x_d. & (x_d : \neg\Diamond\neg A@\mathbf{w}) \\
\quad \texttt{box}\ \omega'. & \text{(need to return } A\text{)} \\
\quad\quad \texttt{letcc}\ u\ \texttt{in}\ \texttt{abort}\ \texttt{get}[\mathbf{w}] & \text{(applying } x_d \text{ will yield } \bot\text{)} \\
\quad\quad\quad x_d(\texttt{get}[\mathbf{w}]\ (\texttt{here}(\lambda a.\texttt{throw}\ a\ \texttt{to}\ u)))
\end{array}$$

In each example, we'll assume that the whole term lives at the constant world $\mathbf{w}$. Operationally, the reading of $\neg\Diamond\neg A \supset \Box A$ is that given a continuation x_d (expecting the address of an A continuation), we will return a boxed A that is well-formed anywhere. The proof term given accomplishes this by creating a box that, when opened, grabs the current continuation u, which has type $A \star \omega'$. With the continuation in hand, we travel back to ω (where x_d lives), and apply x_d to the address of a function that throws to u. In short, at the moment the box is opened we have a *lack of* an A, which we can grab with `letcc` and then take the address of with `here`. This is enough to send to the continuation that we're provided.

Dually we can define $\Diamond$ in terms of $\Box$. Again, one direction is intuitionistically valid. The other,

$$\neg\Box\neg A \supset \Diamond A$$

is a function from a continuation to the address of some arbitrary A. It is implemented by the following proof term:

$$\begin{array}{ll}
\lambda x_c. & (x_c : \neg\Box\neg A@\mathbf{w}) \\
\quad \texttt{letcc}\ u\ \texttt{in} & (u : \Diamond A \star \mathbf{w}) \\
\quad\quad \texttt{abort}(& \text{(applying } x_c \text{ will yield } \bot\text{)} \\
\quad\quad\quad x_c(\texttt{box}\ \omega'.\lambda a. & (a : A@\omega') \\
\quad\quad\quad\quad \texttt{throw}\ (\texttt{get}[\omega'](\texttt{here}\ a))\ \texttt{to}\ u))
\end{array}$$

Here, we immediately do a `letcc`, grabbing the $\Diamond A$ continuation at $\mathbf{w}$. We then form a `box` to pass to the continuation x_c. It contains a function of type $A \supset \bot$, which takes the address of its argument and throws it to the saved continuation u. Thus the source of the $\Diamond A$ that we ultimately return is the world that invokes the continuation (of type $\neg A$) that we've boxed up.

Excluded modal. The last example uses disjunction, which we left out of the calculus. It can be added as a primitive connective in the same way as we did in Lambda 5 (Section 3.2.3), or by de Morgan translation using the $\neg$ and $\wedge$ connectives. In order to do one more informal example, let's assume the existence of proof terms `inl` and `inr` for injecting into disjunctive type. Then we can prove the classical theorem

$$\Box A \vee \Diamond \neg A$$

which is similar to the excluded middle axiom $A \vee \neg A$. The proof term exhibits both the "space travel" (moving between worlds) of modal logic and "time travel" (non-local control flow) of classical logic. It is as follows:

```
letcc u_d in                    (u_d : □A ∨ ◇¬A ⋆ w)
  inl (box ω'.
     letcc u in                 (u : A ⋆ ω')
       throw (inr (get[ω'] here (λa.throw a to u)))
       to u_d)
```

We start by immediately grabbing the current continuation as u_d, so that we can later "change our minds" as to whether the left ($\Box A$) or right ($\Diamond \neg A$) disjunct is the case. We will originally return a $\Box A$, asserting that A holds everywhere. If this is ever unboxed, however, it grabs the current continuation u (which expects A at some world) and uses that to construct a proof of $\Diamond \neg A$, which it then returns to the original continuation u_d. If *this* proof is ever used (by passing A to it to form a contradiction), it returns to the world where the box was opened and can now satisfy the continuation expecting an A!

Having observed its capacity for time and space travel, we now turn our attention to a formal definition of the operational semantics for C5.

3.4.6 Operational semantics

For Lambda 5, I gave an operational semantics based on substitution and a big-step evaluation relation (Section 3.3). This semantics was clean, but does not closely resemble how the programming language ML5 will actually be implemented. The dynamic semantics of C5 will be more realistic in two senses. First, it uses a small-step evaluation relation that relates network states with explicit control stacks. This allows us to see how the state of the network evolves over time, to explain the behavior of infinitely looping programs, and to transition smoothly to a nondeterministic concurrent semantics. Second, some values at runtime cannot really be transmitted between worlds because they

represent local resources, however, the substitution-based Lambda 5 semantics appears to do just this. The C5 semantics uses tables to store local resources, so that integer indexes into these tables can be transmitted between worlds instead. It also uses tables for continuations, so that we do not need to transmit control stacks. In the implementation of ML5, we will use a similar technique to marshal certain values (Section 5.5.4), so C5's table-based semantics provides some of the justification for that. However, because this semantics is awkward to formalize in Twelf (Appendix A.5), this is the last store-based dynamic semantics that we give.

types	A, B	$::=$	$A \supset B \mid \Box A \mid \Diamond A \mid A \wedge B \mid \bot \mid A\,\mathtt{at}\,\mathrm{w}$
networks	$\mathbb{N}$	$::=$	$\mathbb{W}; R$
configs	$\mathbb{W}$	$::=$	$\{\mathbf{w}_1 : \langle \chi_1, b_1 \rangle, \cdots\}$
cursors	R	$::=$	$\mathbf{w} : [k \prec v] \mid \mathbf{w} : [k \succ M]$
tables	b	$::=$	$\bullet \mid b, \ell = v$
cont tables	χ	$::=$	$\bullet \mid \chi, \mathcal{K} = k$
config types	Σ	$::=$	$\{\mathbf{w}_1 : \langle X_1, \beta_1 \rangle, \cdots\}$
table types	β	$::=$	$\bullet \mid \beta, \ell : A$
ctable types	X	$::=$	$\bullet \mid X, \mathcal{K} : A$
world exps	w	$::=$	$\omega \mid \mathbf{w}$
world vars	ω		world names $\mathbf{w}$
labels	ℓ		value vars x, y
cont labs	$\mathcal{K}$		cont vars u
values	v	$::=$	$\lambda x.M \mid \mathtt{box}\ \omega.M \mid \mathbf{w}.\ell \mid \langle v, v' \rangle \mid \mathtt{held}\ v$
conts	k	$::=$	$\mathtt{return}\ Z \mid \mathtt{finish} \mid \mathtt{abort} \mid k \lhd f$
cont exps	Z	$::=$	$\mathbf{w}.\mathcal{K} \mid u$
frames	f	$::=$	$\circ\ N \mid v\ \circ \mid \mathtt{here}\ \circ \mid \mathtt{unbox}\ \circ \mid \mathtt{hold}\ \circ$
			$\mid \mathtt{letd}\ \omega.x = \circ\ \mathtt{in}\ N \mid \pi_n\ \circ \mid \langle \circ, N \rangle \mid \langle v, \circ \rangle$
			$\mid \mathtt{leta}\ x = \circ\ \mathtt{in}\ N$
exps	M, N	$::=$	$v \mid MN \mid x \mid \ell \mid \mathtt{here}\ M \mid \mathtt{get}[\mathrm{w}]\ M$
			$\mid \mathtt{unbox}\ M \mid \mathtt{letd}\ \omega.x = M\ \mathtt{in}\ N$
			$\mid \mathtt{abort}\ M \mid \mathtt{letcc}\ u\ \mathtt{in}\ M \mid \mathtt{hold}\ M$
			$\mid \mathtt{throw}\ M\ \mathtt{to}\ Z \mid \langle M, N \rangle \mid \pi_n\ M$
			$\mid \mathtt{leta}\ x = M\ \mathtt{in}\ N$

Figure 3.15: The syntax for the C5 operational semantics. A network consists of a set of worlds paired with a cursor; each world has a table of values and continuations. A cursor indicates the evaluation state of a single thread at the selected world.

For the C5 operational semantics we give require a number of new syntactic constructs, given in Figure 3.15. The small-step semantics is given in terms of an abstract network that steps from state to state. The network is built out of a fixed number of constant worlds, whose names we write continue to write as bold $\mathbf{w}$. A network state $\mathbb{N}$ has two parts. First is a world configuration $\mathbb{W}$ which associates two tables with each world $\mathbf{w}_i$ present. The first table χ_i stores that world's continuations by mapping continuation

labels $\mathcal{K}$ to literal continuations k. The second table b_i maps value labels ℓ to values in order to store values whose address we have published (with `here`). These tables have types X and β respectively (which map labels $\mathcal{K}$ and ℓ to types), and so we can likewise construct the type of an entire configuration, written Σ.

Aside from the current world configuration, a network state also contains a *cursor* denoting the current focus of computation. The cursor either takes the form $\mathbf{w} : [k \prec v]$ (returning the value v to the continuation k) or $\mathbf{w} : [k \succ M]$ (evaluating the expression M in continuation k). In either case it selects a specific world $\mathbf{w}$ where the computation is taking place.

Continuations themselves are stacks of frames (expressions with a "hole," written $\circ$) with a bottommost `return`, `finish` or `abort`. The `finish` continuation represents the end of computation, so a network state whose cursor is returning a value to `finish` is called *terminal*. The `abort` continuation will be unreachable, and `return` will send the received value to a remote continuation.

Most of the expressions and values are straightforward. As in Lambda 5, the canonical value for $\Box$ abstracts over the hypothetical world and leaves its argument unevaluated (`box` $\omega'.M$). For $\Diamond A$, which represents the address of a value at an undisclosed world, we no longer ship the actual value but an index into some world's table, paired with the name of that world. This value takes the form $\mathbf{w}.\ell$, and is well-formed anywhere (assuming that $\mathbf{w}$'s table has a label ℓ containing a value of type A). On the other hand we have another sort of label, written just ℓ, which is disembodied from its world. These labels arise from the `letd` construct, which deconstructs an address $\mathbf{w}.\ell$ into its components $\mathbf{w}$ and ℓ (see the $\Diamond$ E rule from Figure 3.11). Disembodied labels only make sense at a single world—here ℓ would have type A`@`$\mathbf{w}$. When we attempt to evaluate a disembodied label, we look it up in the current world's table and return the associated value.

Although the external language only allows a `throw` to a continuation variable, intermediate states of evaluation require that these also include the continuation expression $\mathbf{w}.\mathcal{K}$, which pairs a continuation label (an index into the continuation table) with the world at which it lives. These continuation expressions are filled in by `letcc`.

The type system is given in Figure 3.17, omitting the rules that are the same as in Figure 3.11 except for the configuration typing Σ. There are several judgments involved, an index of which appear in Figure 3.16.

The rules addr and lab are used to type the run-time artifacts of address publishing. In either case, we look up the type in the appropriate table typing β, which is part of the configuration type Σ. As mentioned, throw allows a continuation expression Z, which must take the form of a variable (typed with hyp*, as in the logic) or address into a continuation table.

Typing of literal continuations k is straightforward. Note that the judgment $\Sigma \vdash k : A \star \mathrm{w}$ means that the continuation k *expects* a value of type A at w. The `return` continuation arises from `get`, so it allows values of mobile type to be returned. We reuse the network continuation mechanism here to refer to the outstanding `get` on the remote machine.

For an entire network to be well-formed (Rule net), all of the tables must have the

Judgment	Reading
$\Sigma; \Gamma \vdash M : A @ \mathrm{w}$	The expression M has type A at world w
$\Sigma \vdash k : A \star \mathrm{w}$	The continuation k expects a value of type A at world w
$\Sigma; \Gamma \vdash Z : A \star \mathrm{w}$	The continuation expression Z is well-formed, expecting A at w
$\Sigma \vdash b @ \mathbf{w}$	The value table b is well-formed at the world named $\mathbf{w}$
$\Sigma \vdash \chi \star \mathbf{w}$	The continuation table χ is well-formed at the world named $\mathbf{w}$
$\Sigma \vdash R$	The cursor is well-formed
$\Sigma \vdash \mathbb{N}$	The network is well-formed

Figure 3.16: Index of Judgments for the C5 operational semantics. In each judgment Σ is a configuration typing and Γ is a context of value, world and continuation hypotheses.

type indicated by the configuration type Σ, which means that they must have exactly the same labels, and the values or continuations must be well-typed at the specified types (Rules b and χ). Finally, the cursor must be well-formed: It must select a world that exists in the network, and there must exist a type A such that its continuation and value or expression both have type A at that world and are closed.

Having set up the syntax and type system, we can now give the evaluation relation and prove a type safety theorem for it.

Evaluation relation

The operational semantics of C5 is given in Figure 3.18 as a binary relation $\mapsto$ between network states. The semantics evaluates programs sequentially, though we show how the semantics can be made concurrent in the next section.

At any step, the cursor is selecting a world and continuation, with a value to return to it or an expression to evaluate. The rules generally fall into a few categories, as exemplified by the (standard) rules for $\supset$: There are **p**ush rules, in which we begin evaluating a subexpression of some M, pushing the context into the continuation, **s**wap rules, where we have finished evaluating one sub-expression and move onto the next, and **r**eduction rules, where we have a value and consume it. Every well-typed machine state is closed with respect to value, continuation, and world hypotheses, so we don't have rules for variables and can constrain some rules to operate only on constants.

The first interesting rule is $\Diamond_i$-r, which publishes the value v and returns its address. Executing at $\mathbf{w}$, it generates a new label, maps that label to v within $\mathbf{w}$'s private value table, and returns the pair $\mathbf{w}.\ell$. Whenever we try to evaluate a label (Rule ℓ-r), we look it up in the current world's value table in order to fetch the value. A key consequence of type safety (Theorems 21, 22) is that labels are only evaluated in the correct world. To eliminate an address (Rule $\Diamond_e$-r) we substitute the constituent world and label through the body of the `letd`. Note that this step is slightly non-standard, because we substitute the *expression* ℓ for a variable rather than some value. We delay the lookup of the value to the point where ℓ is evaluated (at its home world) so that we only have to look in

$$\frac{\Sigma(\mathbf{w}) = \langle X, \beta\rangle \quad \beta(\ell) = A}{\Sigma;\Gamma \vdash \mathbf{w}.\ell : \Diamond A@\mathrm{w}'}\ \text{addr} \qquad \frac{\Sigma(\mathbf{w}) = \langle X, \beta\rangle \quad \beta(\ell) = A}{\Sigma;\Gamma \vdash \ell : A@\mathbf{w}}\ \text{lab}$$

$$\frac{\Sigma;\Gamma \vdash M : A@\mathrm{w} \quad \Sigma;\Gamma \vdash Z : A\star\mathrm{w}}{\Sigma;\Gamma \vdash \mathtt{throw}\, M\, \mathtt{to}\, Z : C@\mathrm{w}'}\ \text{throw} \qquad \frac{\Sigma;\Gamma, u{:}A\star\mathrm{w} \vdash M : A@\mathrm{w}}{\Sigma;\Gamma \vdash \mathtt{letcc}\, u\, \mathtt{in}\, M : A@\mathrm{w}}\ \text{letcc}$$

$$\frac{\Sigma(\mathbf{w}) = \langle X, \beta\rangle \quad X(\mathcal{K}) = A}{\Sigma;\Gamma \vdash \mathbf{w}.\mathcal{K} : A\star\mathbf{w}}\ \text{addr}^\star \qquad \frac{}{\Sigma;\Gamma, u{:}A\star\mathrm{w} \vdash u : A\star\mathrm{w}}\ \text{hyp}^\star$$

$$\frac{\Sigma;\Gamma \vdash v : A@\mathrm{w}'}{\Sigma;\Gamma \vdash \mathtt{held}\, v : A\, \mathtt{at}\, \mathrm{w}'@\mathrm{w}}\ \text{held} \qquad \frac{}{\Sigma \vdash \mathtt{abort} : \bot\star\mathrm{w}}\ \text{ka} \qquad \frac{}{\Sigma \vdash \mathtt{finish} : A\star\mathrm{w}}\ \text{kf}$$

$$\frac{A\ \mathsf{mobile} \quad \Sigma;\cdot \vdash Z : A\star\mathrm{w}'}{\Sigma \vdash \mathtt{return}\, Z : A\star\mathrm{w}}\ \text{kret} \qquad \frac{\Sigma \vdash k : \Diamond A\star\mathrm{w}}{\Sigma \vdash k \lhd \mathtt{here} \circ : A\star\mathrm{w}}\ \text{khere}$$

$$\frac{\Sigma \vdash k : B\star\mathrm{w} \quad \Sigma;\cdot \vdash N : A@\mathrm{w}}{\Sigma \vdash k \lhd \circ\, N : A \supset B\star\mathrm{w}}\ \text{kapp}_1 \qquad \frac{\Sigma \vdash k : B\star\mathrm{w} \quad \Sigma;\cdot \vdash v : A \supset B@\mathrm{w}}{\Sigma \vdash k \lhd v \circ : A\star\mathrm{w}}\ \text{kapp}_2$$

$$\frac{\Sigma \vdash k : C\star\mathrm{w} \quad \Sigma;\omega\ \mathsf{world}, x{:}A@\omega \vdash N : C@\mathrm{w}}{\Sigma \vdash k \lhd \mathtt{letd}\ \omega.x = \circ\ \mathtt{in}\, N : \Diamond A\star\mathrm{w}}\ \text{kletd} \qquad \frac{\Sigma \vdash k : A\star\mathrm{w}}{\Sigma \vdash k \lhd \mathtt{unbox} \circ : \Box A\star\mathrm{w}}\ \text{kunbox}$$

$$\frac{\Sigma \vdash k : A \wedge B\star\mathrm{w} \quad \Sigma;\cdot \vdash N : B@\mathrm{w}}{\Sigma \vdash k \lhd \langle \circ, N\rangle : A\star\mathrm{w}}\ \text{kpair}_1 \qquad \frac{\Sigma \vdash k : A \wedge B\star\mathrm{w} \quad \Sigma;\cdot \vdash v : A@\mathrm{w}}{\Sigma \vdash k \lhd \langle v, \circ\rangle : B\star\mathrm{w}}\ \text{kpair}_2$$

$$\frac{\Sigma \vdash k : A\, \mathtt{at}\, \mathrm{w}\star\mathrm{w}}{\Sigma \vdash k \lhd \mathtt{hold} \circ : A\star\mathrm{w}}\ \text{khold} \qquad \frac{\Sigma \vdash k : C\star\mathrm{w} \quad \Sigma; x{:}A@\mathrm{w}' \vdash N : C@\mathrm{w}}{\Sigma \vdash k \lhd \mathtt{leta}\, x = \circ\, \mathtt{in}\, N : A\, \mathtt{at}\, \mathrm{w}'\star\mathrm{w}}\ \text{kleta}$$

$$\frac{\beta = (\ell_1 : A_1, \ldots) \quad \Sigma;\cdot \vdash v_1 : A_1@\mathbf{w} \quad \ldots}{\underbrace{\{\cdots, \mathbf{w} : \langle X, \beta\rangle, \cdots\}}_{\Sigma} \vdash \underbrace{\ell_1 = v_1, \ldots}_{b}\ @\mathbf{w}}\ b \qquad \frac{X = (\mathcal{K}_1 : A_1, \ldots) \quad \Sigma \vdash k_1 : A_1\star\mathbf{w} \quad \ldots}{\underbrace{\{\cdots, \mathbf{w} : \langle X, \beta\rangle, \cdots\}}_{\Sigma} \vdash \underbrace{\mathcal{K}_1 = k_1, \ldots}_{\chi}\ \star\, \mathbf{w}}\ \chi$$

$$\frac{\begin{array}{l}\mathbf{w} \in \mathrm{dom}(\Sigma) \\ \Sigma;\cdot \vdash v : A@\mathbf{w} \quad \Sigma \vdash k : A\star\mathbf{w}\end{array}}{\Sigma \vdash \mathbf{w} : [k \prec v]}\ \text{ret} \qquad \frac{\begin{array}{l}\mathbf{w} \in \mathrm{dom}(\Sigma) \\ \Sigma;\cdot \vdash M : A@\mathbf{w} \quad \Sigma \vdash k : A\star\mathbf{w}\end{array}}{\Sigma \vdash \mathbf{w} : [k \succ M]}\ \text{eval}$$

$$\frac{\Sigma \vdash R \quad \Sigma \vdash \chi_i@\mathbf{w}_i \ \ldots \quad \Sigma \vdash b_i@\mathbf{w}_i \ \ldots}{\Sigma \vdash \{\mathbf{w}_1 : \langle \chi_1, b_1\rangle, \cdots, \mathbf{w}_m : \langle \chi_m, b_m\rangle\}; R}\ \text{net}$$

Figure 3.17: The C5 type system, with rules for typing network states, continuation expressions, tables, cursors, etc.

$\supset_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ MN]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \circ N \succ M]$	
$\supset_e$-s	$\mathbb{W}; \mathbf{w} : [k \lhd \circ N \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd v \circ \succ N]$	
$\supset_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd (\lambda x.M)\circ \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [v/x]M]$	
$\wedge_i$-p	$\mathbb{W}; \mathbf{w} : [k \succ \langle M, N\rangle]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \langle \circ, N\rangle \succ M]$	
$\wedge_i$-s	$\mathbb{W}; \mathbf{w} : [k \lhd \langle \circ, N\rangle \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \langle v, \circ\rangle \succ N]$	
$\wedge_i$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \langle v_1, \circ\rangle \prec v_2]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \langle v_1, v_2\rangle]$	
$\wedge_{e_n}$-p	$\mathbb{W}; \mathbf{w} : [k \succ \pi_n M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \pi_n \circ \succ M]$	
$\wedge_{e_n}$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \pi_n \circ \prec \langle v_1, v_2\rangle]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec v_n]$	
$\Box_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{box}\ \omega.M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \mathtt{box}\ \omega.M]$	
$\Diamond_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \mathbf{w}'.\ell]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \mathbf{w}'.\ell]$	
$\supset_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \lambda x.M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \lambda x.M]$	
$\wedge_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \langle v_1, v_2\rangle]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \langle v_1, v_2\rangle]$	
$\mathtt{at}_i$-v	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{held}\ v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \mathtt{held}\ v]$	
$\bot_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{abort}\ M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [\mathtt{abort} \succ M]$	
$\mathtt{at}_i$-p	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{hold}\ M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{hold} \circ \succ M]$	
$\mathtt{at}_i$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{hold} \circ \prec v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \prec \mathtt{held}\ v]$	
$\mathtt{at}_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{leta}\ x = M\ \mathtt{in}\ N]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{leta}\ x = \circ\ \mathtt{in}\ N \succ M]$	
$\mathtt{at}_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{leta}\ x = \circ\ \mathtt{in}\ N \prec \mathtt{held}\ v]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [v/x]N]$	
$\Diamond_i$-p	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{here}\ M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{here} \circ \succ M]$	
$\Diamond_i$-r	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \lhd \mathtt{here} \circ \prec v]$	$\mapsto$	$\{\mathbf{w} : \langle \chi, (b, \ell = v)\rangle, \cdots\}; \mathbf{w} : [k \prec \mathbf{w}.\ell]$	(ℓ fresh)
ℓ-r	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \succ \ell]$	$\mapsto$	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \prec v]$	($b(\ell) = v$)
$\Diamond_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{letd}\ \omega.x = M\ \mathtt{in}\ N]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{letd}\ \omega.x = \circ\ \mathtt{in}\ N \succ M]$	
$\Diamond_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{letd}\ \omega.x = \circ\ \mathtt{in}\ N \prec \mathbf{w}'.\ell]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [\ell/x][\mathbf{w}'/\omega]N]$	
$\Box_e$-p	$\mathbb{W}; \mathbf{w} : [k \succ \mathtt{unbox}\ M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{unbox} \circ \succ M]$	
$\Box_e$-r	$\mathbb{W}; \mathbf{w} : [k \lhd \mathtt{unbox} \circ \prec \mathtt{box}\ \omega.M]$	$\mapsto$	$\mathbb{W}; \mathbf{w} : [k \succ [\mathbf{w}/\omega]M]$	
letcc	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \succ \mathtt{letcc}\ u\ \mathtt{in}\ M]$	$\mapsto$	$\{\mathbf{w} : \langle (\chi, \mathcal{K} = k), b\rangle, \cdots\}; \mathbf{w} : [k \succ [\mathbf{w}.\mathcal{K}/u]M]$	($\mathcal{K}$ fresh)
throw	$\{\mathbf{w}' : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \succ \mathtt{throw}\ M\ \mathtt{to}\ \mathbf{w}'.\mathcal{K}]$	$\mapsto$	$\{\mathbf{w}' : \langle \chi, b\rangle, \cdots\}; \mathbf{w}' : [k' \succ M]$	($\chi(\mathcal{K}) = k'$)
get	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \succ \mathtt{get}[\mathbf{w}']\ M]$	$\mapsto$	$\{\mathbf{w} : \langle (\chi, \mathcal{K} = k), b\rangle, \cdots\}; \mathbf{w}' : [\mathtt{return}\ \mathbf{w}.\mathcal{K} \succ M]$	($\mathcal{K}$ fresh)
ret	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w}' : [\mathtt{return}\ \mathbf{w}.\mathcal{K} \prec v]$	$\mapsto$	$\{\mathbf{w} : \langle \chi, b\rangle, \cdots\}; \mathbf{w} : [k \prec v]$	($\chi(\mathcal{K}) = k$)

Figure 3.18: The small-step evaluation relation $\mapsto$ for C5.

tables locally.

The rules for $\Box$ are much simpler: `box` $\omega.M$ is already a value (Rule $\Box_i$-v), and to `unbox` we simply substitute the current world for the hypothetical one (Rule $\Box_e$-r).

When encountering a `letcc`, we grab the current continuation k. Because the continuation may be referred to from elsewhere in the network, we publish it in a table and form a global address for it (of the form $\mathbf{w}.\mathcal{K}$), just as we did for $\Diamond$ addresses. This address is substituted for the falsehood variable u using standard substitution—*not* the special falsehood substitution (Theorem 15) we used in Section 3.4.2. The latter was a proof-theoretic notion used to eliminate uses of the hypothesis; here we want the use of the hypothesis (`throw`) to have run-time significance. A point of comparison is the above paragraph, where we substituted the expression ℓ for a variable because we wanted to delay the operation until the time the variable is "looked up."

Throwing to a continuation (Rule throw) is handled straightforwardly. The continuation expression will be closed, and therefore of the form $\mathbf{w}'.\mathcal{K}$. We look up the label $\mathcal{K}$ in $\mathbf{w}'$—or rather, *cause* $\mathbf{w}'$ to look it up—and pass the expression M to it. Note that we do not evaluate the argument before throwing it to the remote continuation. In general we *can not* evaluate it, because it is only well-typed at the remote world, which may be different from the world we're in.

Finally, the `get` construct works as follows. Since the target world expression must be closed it will be a world constant in the domain of $\mathbb{W}$. We will move the cursor to that world and begin evaluating the expression M. To arrange for the result of M to be returned to us, we insert our current continuation in the local continuation table. The bottom frame at the remote world is then a `return` to that continuation label. The `return` frame reduces like `throw`. Unlike `throw`, its argument (restricted to mobile types) will be eagerly evaluated—the whole point is to create the value at one world and then move it to another.

Type safety

In order for C5 to make sense it must be type safe: any well-typed program must have a meaning, given as a sequence of steps in the abstract network. For the small-step semantics, we state type safety in terms of progress and preservation:

Theorem 21 (C5 Progress)

If	$\Sigma \vdash \mathbb{N}$	
then	either	$\mathbb{N}$ is terminal
	or	$\exists \mathbb{N}'.\mathbb{N} \mapsto \mathbb{N}'$

Theorem 22 (C5 Preservation)

If	$\Sigma \vdash \mathbb{N}$
and	$\mathbb{N} \mapsto \mathbb{N}'$
then	$\exists \Sigma'.\ \Sigma' \supseteq \Sigma$
such that	$\Sigma' \vdash \mathbb{N}'$

Progress says that any well-formed network state can take another step, or is done. (Recall that a *terminal* network is one where the cursor is returning a value to a `finish`

continuation.) Preservation says that any well-typed network state that takes a step results in another well-typed state (perhaps in an extended configuration typing Σ'. ($\Sigma' \supseteq \Sigma$ iff Σ' and Σ each describe the same set of worlds, and for each world, if $X'(\mathcal{K}) = A$ then $X(\mathcal{K}) = A$, and likewise for β' and β). By iterating alternate applications of these theorems we see that any well-typed program is able to step repeatedly and remain well-formed, or else eventually comes to rest in a terminal state.

The proof of Theorem 21 is by induction over the derivation that $\mathbb{N}$ is well-formed, and proof of Theorem 22 is by induction over the derivation that $\mathbb{N} \mapsto \mathbb{N}'$. These proofs are somewhat difficult to perform in Twelf due to the value/continuation tables; because they are indexed by labels (and not variables) we cannot use Twelf's facilities for higher order abstract syntax to encode them. This means that we must prove substitution and weakening theorems by hand. The full proofs appear in Appendix A.5 as the relations `progress` and `preservation`. □

Uses of continuations

This cursor-based style of operational semantics admits an easy extension to support concurrency. We simply replace the cursor R in our network state $(\mathbb{W}; R)$ with a multiset of cursors $\Re$, each one represent an independent thread. We then permit a step on any one of these cursors essentially according to the old rules. Formally,

$$\begin{array}{ll} & \mathbb{W}; \Re \quad \mapsto^c \quad \mathbb{W}'; \Re' \\ \text{iff} & \Re = R \uplus \Re_{\text{rest}} \\ \text{and} & \mathbb{W}; R \mapsto \mathbb{W}'; R' \\ \text{and} & \Re' = R' \uplus \Re_{\text{rest}} \end{array}$$

We can then add primitives as desired to spawn new cursors. A very simple one evaluates M and N in parallel and returns each one to the same continuation:

$$\frac{\Gamma \vdash M : A@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}}{\Gamma \vdash M|N : A@\mathrm{w}} \text{ par}$$

$$\mathbb{W}; \Re \uplus \mathbf{w}{:}[k \succ M|N] \mapsto^c \mathbb{W}; \Re \uplus \mathbf{w}{:}[k \succ M] \uplus \mathbf{w}{:}[k \succ N]$$

It is easy to see that suitable extensions of progress and preservation hold for $\mapsto^c$.

In ML5, we implement concurrency in the same way. In interactive applications, threads are inserted into the pool as a result of actions by the user such as keypresses and mouseclicks on GUI elements. The interface to the database also allows a hook to be registered with a key, so that the hook is launched as a new thread whenever the key is modified.

This concludes the discussion of C5, a classical modal logic for distributed computing. We will use continuations again starting in Chapter 4 when we formalize the compilation of Lambda 5 via CPS conversion. ML5 also has the `letcc` and `throw` constructs, although there they will be limited to (spatially) local control flow (Chapter 5). In the next section, we will look at an orthogonal extension to Lambda 5 for universal reasoning.

3.5 Validity

In our modal calculi so far, reasoning has always been attached to a specific world via the "true $@\mathrm{w}$" (or "false $\star\,\mathrm{w}$") judgments. This is the critical technology in this work because it allows us to express contingent truths, which correspond to programs that are limited in where they may be executed. Nonetheless, distributed programs usually contain a large fraction of data and code that can be used anywhere: simple data like integers, and library code that manipulates data structures like lists and trees, etc. We call such unconstrained data and code "valid." For these parts of the program, the modal typing discipline is a burden, because it requires us to explicitly move valid things from world to world. For example, the following Lambda 5 program of $\texttt{int} \supset \Box\texttt{int@w}$ is not as direct as we would like:

$$\lambda x{:}\texttt{int}.\ \texttt{box}\,\omega.\texttt{get}[\mathbf{w}]\ x$$

The problem is that the box that we create must return to w every time it is unboxed, in order to retrieve x. This is more work than should be necessary; the box ought to simply contain the integer. Because of the modal discipline, however, there is no way for us to make x available for use at the hypothetical world ω, because we cannot mention the bound world variable until we introduce it with the `box` construct. This turns out to be particularly troublesome when we introduce run-time addresses for worlds in Section 4.2.1. When we begin writing the body of a `box` at some world where truly nothing is known, then we won't even have the ability to `get` data from other worlds, since we will not have access to their run-time addresses!

The solution is to introduce universal reasoning into the calculus. This will allow us to assert that a piece of code or data is valid and can be used anywhere—including hypothetical worlds that are introduced later. We will make such assertions in terms of a new validity judgment, and introduce a modality that internalizes this judgment as a type.

In this development we will see a widening gap between the logic and the programming language based on it. Specifically, we will make a syntactic constraint similar to ML's value restriction [77] on polymorphism so that we always have a specific concrete place in which we are performing evaluation. This leads to an incompleteness of the call-by-value dynamic semantics with respect to the proof theory. (Like ML's value restriction, this is rarely a problem in practice, because we can almost always lambda-abstract something to make it a value.)

To this end, we only form hypotheses of validity, never conclusions. Operationally, a hypothesis tells us that we have a *value* and tells us where we may use that value—a valid hypothesis simply says that we can use it anywhere. The hypothesis form is

$$u \sim \omega.A$$

where u is a new syntactic class of variable. The world variable ω is bound within A and stands for the world(s) at which u is used. It rarely occurs, so we leave it out when it does not (or can not). Valid variables can be used at any world to conclude modal truth at that world (Rule vhyp; Figure 3.19).

$$\frac{}{\Gamma, u \sim \omega.A \vdash u : [^{\mathrm{w}}/_{\omega}]A@\mathrm{w}}\ \text{vhyp} \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : A@\mathrm{w} \quad \Gamma, u \sim A \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{put}\ u = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \text{put}$$

$$\frac{\Gamma, \omega\ \mathsf{world} \vdash M : A@\omega}{\Gamma \vdash \mathtt{sham}\ \omega.M : \mathfrak{Z}_{\omega}A@\mathrm{w}}\ \mathfrak{Z}\,\mathrm{I} \qquad \frac{\Gamma \vdash M : \mathfrak{Z}_{\omega}A@\mathrm{w} \quad \Gamma, u \sim \omega.A \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{letsham}\ u = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \mathfrak{Z}\,\mathrm{E}$$

$$\frac{}{\mathfrak{Z}_{\omega}A\ \mathsf{mobile}}\ \mathfrak{Z}\,\mathrm{M}$$

Figure 3.19: Extensions to Lambda 5 for validity.

A hypothesis should have a corresponding substitution principle, which is as follows:

Theorem 23 (Validity substitution)
If $\Gamma \vdash M : A@\omega$ *(for all ω)*
and $\Gamma, u \sim \omega.A \vdash N : B@\mathrm{w}$
then $\Gamma \vdash [\![^{\omega.M}/_{u}]\!]_{\mathrm{w}} N : B@\mathrm{w}$

The substitution operation $[\![^{\omega.M}/_{u}]\!]_{\mathrm{w}} N$ is indexed by the world at which N is typed, and is defined by induction over N. It is pointwise except for the case for valid variables:

$$[\![^{\omega.M}/_{u}]\!]_{\mathrm{w}} u = [^{\mathrm{w}}/_{\omega}]M$$

That is, when we arrive at a use of the valid variable u, we replace it with the expression M but fill in the world variable ω with the current world w. At runtime we will only substitute values, just as we do with normal substitution in a call-by-value language.

One way to introduce a valid hypothesis is with the `put` construct. For example, the proof term

$$\mathtt{put}\ u = 2 + 3\ \mathtt{in}\ M$$

binds $u \sim \mathtt{int}$ within M. The rule for `put` appears in Figure 3.19. Like `get`, it requires its argument to be mobile. Unlike `get`, it does not cause any communication to occur; it simply binds the variable such that it can be used in any world.

The $\mathfrak{Z}$ modality (pronounced "shamrock") is the internalization of the validity judgment as a type. Operationally, a value of type $\mathfrak{Z}A$ is an encapsulated value of type A that makes sense at any world. (Again the hypothetical world can appear in the type; when it does we write $\mathfrak{Z}_{\omega}A$.) Its introduction rule is the same as the rule for $\Box$ in the proof theory, except for the possible appearance of the bound world variable. (In the dynamic semantics (Section 3.5.2) we will require the body of the shamrock to be a value.) The elimination rule $\mathfrak{Z}$ E binds a valid variable in the body of the `letsham`.

Like the other modalities, $\mathfrak{Z}A$ is mobile for any A.

3.5.1 Sequent calculus

To show that this natural deduction makes sense, we prove it is equivalent to a sequent calculus with a cut principle and the subformula property. We do this for a miniature

$$\frac{\Gamma, A \supset B @ \mathrm{w} \Longrightarrow A @ \mathrm{w} \quad \Gamma, A \supset B @ \mathrm{w}, B @ \mathrm{w} \Longrightarrow C @ \mathrm{w}'}{\Gamma, A \supset B @ \mathrm{w} \Longrightarrow C @ \mathrm{w}'} \supset \mathrm{L} \qquad \frac{\Gamma, A @ \mathrm{w} \Longrightarrow B @ \mathrm{w}}{\Gamma \Longrightarrow A \supset B @ \mathrm{w}} \supset \mathrm{R}$$

$$\frac{\Gamma, A \,\mathtt{at}\, \mathrm{w} @ \mathrm{w}', A @ \mathrm{w} \Longrightarrow C @ \mathrm{w}''}{\Gamma, A \,\mathtt{at}\, \mathrm{w} @ \mathrm{w}' \Longrightarrow C @ \mathrm{w}''} \,\mathtt{at}\, \mathrm{L} \qquad \frac{\Gamma \Longrightarrow A @ \mathrm{w}}{\Gamma \Longrightarrow A \,\mathtt{at}\, \mathrm{w} @ \mathrm{w}'} \,\mathtt{at}\, \mathrm{R}$$

$$\frac{\Gamma, \mathfrak{B}_\omega A @ \mathrm{w}', \sim \omega.A \Longrightarrow C @ \mathrm{w}}{\Gamma, \mathfrak{B}_\omega A @ \mathrm{w}' \Longrightarrow C @ \mathrm{w}} \mathfrak{B}\,\mathrm{L} \qquad \frac{\Gamma, \omega\ \mathsf{world} \Longrightarrow A @ \omega}{\Gamma \Longrightarrow \mathfrak{B}_\omega A @ \mathrm{w}} \mathfrak{B}\,\mathrm{R}$$

$$\frac{}{\Gamma \Longrightarrow \top @ \mathrm{w}} \top \mathrm{R} \qquad \frac{}{\Gamma, \bot @ \mathrm{w} \Longrightarrow C @ \mathrm{w}'} \bot \mathrm{L}$$

$$\frac{\Gamma, \sim \omega.A, [{}^{\mathrm{w}'}/_{\omega}]A @ \mathrm{w}' \Longrightarrow C @ \mathrm{w}}{\Gamma, \sim \omega.A \Longrightarrow C @ \mathrm{w}} \text{ copy} \qquad \frac{}{\Gamma, A @ \mathrm{w} \Longrightarrow A @ \mathrm{w}} \text{ init}$$

Figure 3.20: A miniature sequent calculus with validity and the $\mathfrak{B}$ modality. Here, hypotheses are not labeled with variable names, so a valid hypothesis takes the form "$\sim \omega.A$".

system with very few connectives, since the reasoning for the remainder would be the same as it was for Lambda 5. The sequent calculus appears in Figure 3.20.

The rules for $\supset$ and `at` are the same as before, as is the initial sequent. The rules for the $\mathfrak{B}$ modality are straightforward; the right rule mimics the introduction rule from natural deduction. The left rule produces a validity hypothesis. To use a validity hypothesis, we choose a world and apply the copy rule to create a modal hypothesis of the same formula at that world (with the target world substituted in for the hypothetical one). This is analogous to the copy rule used in Chang, Chaudhuri, and Pfenning's sequent calculus for linear logic [17]. After creating a modal hypothesis it can be used like any other with the init rule.

Like the previous two sequent calculi, this one has the subformula property and so it is easy to see consistency. We also have shift and expansion lemmas:

Theorem 24 (Consistency)
Not all sequents are provable.

Theorem 25 (Expansion)
If A mobile
then $A @ \mathrm{w} \Longrightarrow A @ \mathrm{w}'$

Theorem 26 (Shift)
If A mobile
and $\Gamma \Longrightarrow A @ \mathrm{w}$
then $\Gamma \Longrightarrow A @ \mathrm{w}'$

Proof of Theorem 24 is immediate by witness (no rule can conclude $\cdot \Longrightarrow \bot @ \mathbf{w}$), and Theorems 25 and 26 proceed as Theorems 3 and 4 for Lambda 5, extended straightfor-

wardly to the new $\fatbslash$ connective. These appear as the Twelf relations `shift` and `exp` in Appendix A.6. □

For this sequent calculus there are two cut principles, one for each form of hypothesis:

Theorem 27 (Cut)
If $\mathcal{D} :: \Gamma, \Gamma' \Longrightarrow A@\mathrm{w}$
and $\mathcal{E} :: \Gamma, A@\mathrm{w}, \Gamma' \Longrightarrow B@\mathrm{w}'$
then $\mathcal{F} :: \Gamma, \Gamma' \Longrightarrow B@\mathrm{w}'$

Theorem 28 (Valid cut)
If $\mathcal{D} :: \Gamma, \Gamma' \Longrightarrow A@\omega$ *(for all ω)*
and $\mathcal{E} :: \Gamma, \sim\omega.A, \Gamma' \Longrightarrow B@\mathrm{w}'$
then $\mathcal{F} :: \Gamma, \Gamma' \Longrightarrow B@\mathrm{w}'$

Proof of Theorems 27 and 28 is by mutual lexicographic induction over the size of the cut formula A, then simultaneously the derivations $\mathcal{D}$ and $\mathcal{E}$. We define the size of A such that the parameterized proposition $\omega.A$ is larger than $[{}^{\mathrm{w}}/_{\omega}]A$ for any w. This relies on worlds not having any inductive structure. The interesting case for Cut is the principal cut for the $\fatbslash$ connective. If

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ \Gamma, \omega\ \mathsf{world} \Longrightarrow A@\omega\end{array}}{\Gamma \Longrightarrow \fatbslash_\omega A@\mathrm{w}}\ \fatbslash\,\mathrm{R} \qquad \text{and } \mathcal{E} \quad = \quad \dfrac{\begin{array}{c}\mathcal{E}' \\ \vdots \\ \Gamma, \fatbslash_\omega A@\mathrm{w}', \sim\omega.A \Longrightarrow C@\mathrm{w}\end{array}}{\Gamma, \fatbslash_\omega A@\mathrm{w}' \Longrightarrow C@\mathrm{w}}\ \fatbslash\,\mathrm{L}$$

then by induction hypothesis on $\mathcal{D}$ and $\mathcal{E}'$ we have

$$\mathcal{E} :: \Gamma, \sim\omega.A \Longrightarrow C@\mathrm{w}$$

then by inductive appeal to Valid cut on $\mathcal{D}'$ and $\mathcal{E}'$ (justified by the smaller cut formula) we have

$$\mathcal{F} :: \Gamma \Longrightarrow C@\mathrm{w}$$

as required.

The cases for Valid cut generally mimic the ones for Cut; there are no principal cases since none of the left rules can operate directly on a valid hypothesis. Therefore, we also never need to distinguish cases on the rule that concludes $\mathcal{D}$. The interesting case is when we copy the cut formula; if

$$\mathcal{E} \quad = \quad \dfrac{\begin{array}{c}\mathcal{E}' \\ \vdots \\ \Gamma, \sim\omega.A, [{}^{\mathrm{w}'}/_{\omega}]A@\mathrm{w}' \Longrightarrow C@\mathrm{w}\end{array}}{\Gamma, \sim\omega.A \Longrightarrow C@\mathrm{w}}\ \text{copy}$$

then we have by induction hypothesis on $\mathcal{D}$ and $\mathcal{E}'$

$$\mathcal{E}'' :: \Gamma, [{}^{\mathrm{w}'}/_{\omega}]A@\mathrm{w}' \Longrightarrow C@\mathrm{w}$$

To eliminate this hypothesis, we first observe that since $\mathcal{D}$ is parametric in its world,

$$[^{\mathrm{w}'}/_{\omega}]\mathcal{D} :: \Gamma \Longrightarrow [^{\mathrm{w}'}/_{\omega}]A@\mathrm{w}'$$

and so by an appeal back to Cut on $[^{\mathrm{w}'}/_{\omega}]\mathcal{D}$ and $\mathcal{E}'$ (justified by the fact that $[^{\mathrm{w}'}/_{\omega}]A$ is smaller than $\omega.A$ by our metric) we have

$$\mathcal{F} :: \Gamma \Longrightarrow C@\mathrm{w}$$

as required. The full proof appears as the relations `cut` and `vcut` in Appendix A.6. □

Given Cut, we can prove soundness and completeness in the typical way:

Theorem 29 (Soundness of validity extensions)
If $\mathcal{D} :: \Gamma \vdash M : A@\mathrm{w}$
then $\mathcal{E} :: \ulcorner\Gamma\urcorner \Longrightarrow A@\mathrm{w}$

Theorem 30 (Completeness of validity extensions)
If $\mathcal{D} :: \Gamma \Longrightarrow A@\mathrm{w}$
then $\mathcal{E} :: \Gamma' \vdash M : A@\mathrm{w}$, for some M and $\Gamma' \sqsupset \Gamma$

These proofs follow the same structure as Theorems 5 and 6 for Lambda 5. It is interesting to note that we completely eliminate the `put` construct when translating to the sequent calculus and do not use it when translating back to natural deduction; therefore it is an admissible rule. The full proofs appear in Appendix A.6 as the relations `ndseq`, `uhyp`, `seqnd`, `hypnd`, and `vhypnd`. □

3.5.2 Operational semantics

For the purpose of the operational semantics, we make a restriction on the introduction form for ⟀: that the body be a syntactic value. To see why, assume that we have ML-style references in our language and consider the following program (typed at the world `home`):

```
letsham u = sham wv. ref 0
in
   u := 1;
   get[w'] (! u)
end
```

Our options for evaluating this program are as follows. We could delay evaluation of the body of the `sham` and substitute the *expression* `ref 0` through the body. The result is that two different reference cells are created—one for each use—at both `home` and `w'`. This not what we would expect from a call-by-value language. We could attempt to evaluate the body of the `sham` once. If we do, then we create a reference cell at the hypothetical world ω and then attempt to access it at both `home` and `w'`, which is unsound. Instead, we reject this program because `ref 0` is not a value. This is the same value restriction we have on type polymorphism in ML, which is not surprising because ⟀ is used for a kind of polymorphism in ML5 (Section 5.3.3). When we do this, we will actually relax the restriction to *valuable* (total, side-effect free) expressions, like ML.

$$\text{values} \quad v \quad ::= \quad \boldsymbol{\lambda} x.M \mid \mathbf{held}\, v \mid \mathbf{sham}\, \omega.v \mid x \mid u$$

$$\frac{}{\Gamma, x{:}A@\mathrm{w} \vdash x : A@\mathrm{w}}\ \text{hyp} \qquad \frac{}{\Gamma, u \sim \omega.A \vdash u : [{}^{\mathrm{w}}/_{\omega}]A@\mathrm{w}}\ \text{vhyp}$$

$$\frac{\Gamma, x{:}A@\mathrm{w} \vdash M : B@\mathrm{w}}{\Gamma \vdash \boldsymbol{\lambda} x.M : A \supset B@\mathrm{w}}\ \supset\ \mathrm{V} \qquad \frac{\Gamma \vdash M : A \supset B@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}}{\Gamma \vdash M\ N : B@\mathrm{w}}\ \supset\ \mathrm{E}$$

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{hold}\, M : A\, \mathtt{at}\, \mathrm{w}@\mathrm{w}'}\ \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma \vdash M : A\, \mathtt{at}\, \mathrm{w}''@\mathrm{w}' \quad \Gamma, x{:}A@\mathrm{w}'' \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{leta}\, x = M\, \mathtt{in}\, N : C@\mathrm{w}}\ \mathtt{at}\ \mathrm{E}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}'}{\Gamma \vdash \mathbf{held}\, v : A\, \mathtt{at}\, \mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{V} \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\omega}{\Gamma \vdash \mathbf{sham}\, \omega.v : \mathfrak{B}_{\omega} A@\mathrm{w}}\ \mathfrak{B}\ \mathrm{V}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathtt{val}\, v : A@\mathrm{w}}\ \text{val} \qquad \frac{\Gamma \vdash M : \mathfrak{B}_{\omega} A@\mathrm{w} \quad \Gamma, u \sim \omega.A \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{letsham}\, u = M\, \mathtt{in}\, N : C@\mathrm{w}}\ \mathfrak{B}\ \mathrm{E}$$

$$\frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : A@\mathrm{w} \quad \Gamma, u \sim A \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{put}\, u = M\, \mathtt{in}\, N : C@\mathrm{w}}\ \text{put} \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash M : A@\mathrm{w}'}{\Gamma \vdash \mathtt{get}[\mathrm{w}']\, M : A@\mathrm{w}}\ \text{get}$$

Figure 3.21: A miniature type system exhibiting the validity extensions.

Unfortunately, disallowing expressions as the body of the `sham` construct makes it incomplete with respect to the sequent calculus. The language of course remains sound, because values are a strict subset of expressions.

The full type system appears in Figure 3.21. Values now include valid variables u and $\mathtt{sham}\, \omega.v$. As discussed, the body of **sham** is limited to a syntactic value.

As for Lambda 5, I will give the operational semantics in terms of a big-step evaluation relation $\Downarrow$ that relates an expression with a value. Its definition appears in Figure 3.22. As before, the relation is indexed by the world in which evaluation is taking place, and this world is always concrete (a constant). The two new rules are put $\Downarrow$ and letsham $\Downarrow$. The `put` construct evaluates its argument to a value. It then uses validity substitution (Theorem 23) to replace the valid variable from the body. Because the value v produced by evaluating the argument M must be closed, the world variable ω bound in validity substitution can not occur, so this abstraction is vacuous. The `letsham` construct works similarly. It evaluates its argument to a value of the form $\mathbf{sham}\ \omega.v$, and then uses validity substitution on that value to substitute away the bound variable in the body.

$$\frac{M \Downarrow_{\mathbf{w}} \boldsymbol{\lambda} x.M' \quad N \Downarrow_{\mathbf{w}} v' \quad [{}^{v'}/_{x}]M' \Downarrow_{\mathbf{w}} v}{M\ N \Downarrow_{\mathbf{w}} v}\ \text{app}\ \Downarrow$$

$$\frac{M \Downarrow_{\mathbf{w}} v}{\mathtt{hold}\ M \Downarrow_{\mathbf{w}} \mathbf{held}\ v}\ \text{hold}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \mathbf{held}\ v' \quad [{}^{v'}/_{x}]N \Downarrow_{\mathbf{w}} v}{\mathtt{leta}\ x = M\ \mathtt{in}\ N \Downarrow_{\mathbf{w}} v}\ \text{leta}\ \Downarrow$$

$$\frac{}{\mathtt{val}\ v \Downarrow_{\mathbf{w}} v}\ \text{val}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} \mathbf{sham}\ \omega.v \quad [\![{}^{\omega.v}/_{u}]\!]_{\mathbf{w}} N \Downarrow_{\mathbf{w}} v'}{\mathtt{letsham}\ u = M\ \mathtt{in}\ N \Downarrow_{\mathbf{w}} v'}\ \text{letsham}\ \Downarrow$$

$$\frac{M \Downarrow_{\mathbf{w}'} v}{\mathtt{get}[\mathbf{w}']\ M \Downarrow_{\mathbf{w}} v}\ \text{get}\ \Downarrow \qquad \frac{M \Downarrow_{\mathbf{w}} v \quad [\![{}^{\omega.v}/_{u}]\!]_{\mathbf{w}} N \Downarrow_{\mathbf{w}} v'}{\mathtt{put}\ u = M\ \mathtt{in}\ N \Downarrow_{\mathbf{w}} v'}\ \text{put}\ \Downarrow$$

Figure 3.22: Big-step dynamic semantics for the miniature validity calculus, given as a relation $M \Downarrow_{\mathbf{w}} v$ between an expression M and its value v.

3.5.3 Type safety

To prove type safety, we need the validity substitution theorem (Theorem 23) and a value shift lemma:

Theorem 31 (Value shift)
If ω_1 world, $\ldots \omega_n$ world $\vdash v : A@\mathrm{w}$
and A mobile
then ω_1 world, $\ldots \omega_n$ world $\vdash v : A@\mathrm{w}'$

This lemma is proved the same way as it was for Lambda 5 (Theorem 23), but here we extend the theorem statement from the empty context to a context that may contain hypothetical worlds. This stronger form is used for the case of `put` below. Its proof appears as the relation `vsubst` in Appendix A.7. □

Type preservation is stated as follows:

Theorem 32 (Type preservation of $\Downarrow$)
If $\mathcal{D} :: \cdot \vdash M : A@\mathbf{w}$
and $\mathcal{E} :: M \Downarrow_{\mathbf{w}} v$
then $\mathcal{F} :: \cdot \vdash v : A@\mathbf{w}$

The proof of this theorem is by induction on $\mathcal{E}$. The case for `letsham` is interesting; if

$$\mathcal{E} \quad = \quad \frac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \vdots & \vdots \\ M \Downarrow_{\mathbf{w}} \mathbf{sham}\ \omega.v & [\![{}^{\omega.v}/_{u}]\!]_{\mathbf{w}} N \Downarrow_{\mathbf{w}} v'\end{array}}{\mathtt{letsham}\ u = M\ \mathtt{in}\ N \Downarrow_{\mathbf{w}} v'}\ \text{letsham}\ \Downarrow$$

then by inversion on $\mathcal{D}$ we have

$$\mathcal{D}_1 :: \cdot \vdash M : \text{⟨⟩}_{\omega} A@\mathbf{w} \qquad \text{and} \qquad \mathcal{D}_2 :: u \sim \omega.A \vdash N : C@\mathbf{w}$$

so by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{E}_1$ we get

$$\mathcal{E}_1' :: \cdot \vdash \mathbf{sham}\ \omega.v : \text{⟨⟩}_{\omega} A@\mathbf{w}$$

by inversion on this we have

$$\mathcal{E}_1'' :: \omega \text{ world} \vdash v : A@\omega$$

so by valid substitution on $\mathcal{E}_1''$ and $\mathcal{D}_2$ we get

$$\mathcal{F}_2 :: \cdot \vdash [^{\omega.v}/_u]_{\mathbf{w}} N : C@\mathbf{w}$$

and then by induction hypothesis on $\mathcal{F}_2$ and $\mathcal{E}_2$ we get

$$\mathcal{F} :: \cdot \vdash v' : C@\mathbf{w}$$

as required.

The case for `put` is similar but more subtle because we must derive the validity of a value from the mobility of its type. Our strengthened value shift lemma allows us to do this. If

$$\mathcal{E} \quad = \quad \frac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \vdots & \vdots \\ M \Downarrow_{\mathbf{w}} v & [^{\omega.v}/_u]_{\mathbf{w}} N \Downarrow_{\mathbf{w}} v'\end{array}}{\texttt{put}\ u = M\ \texttt{in}\ N \Downarrow_{\mathbf{w}} v'}\ \text{put}\ \Downarrow$$

then by inversion on $\mathcal{D}$ we have

$$\mathcal{D}_1 :: A \text{ mobile} \qquad \text{and} \qquad \mathcal{D}_2 :: \cdot \vdash M : A@\mathbf{w} \qquad \text{and} \qquad \mathcal{D}_3 :: u \sim A \vdash N : C@\mathbf{w}$$

so we get by induction hypothesis

$$\mathcal{E}_1' :: \cdot \vdash v : A@\mathbf{w}$$

as before. To use valid substitution, we need that v is well typed at *any* world. We can use the stronger value shift lemma to do this. We first weaken $\mathcal{E}_1'$ to include a hypothetical world in its context, giving

$$\mathcal{E}_1'' :: \omega \text{ world} \vdash v : A@\mathbf{w}$$

and then apply value shift, with the destination being that hypothetical world, giving

$$\mathcal{E}_1''' :: \omega \text{ world} \vdash v : A@\omega$$

we then proceed using valid substitution and induction as above.

The full proof appears as the type correctness of the relation `eval` in Appendix A.7, along with the lemmas `vsubst` and `vshift`. The proof requires a trick which may be interesting to Twelf experts; I comment on it briefly in the appendix. □

We also desire a progress result for the evaluation relation; this takes the form of the coverage (or partiality) of the evaluation relation as encoded in LF, as before. □

3.5.4 Relationship with other connectives

The $\mathfrak{B}$ connective is related to—but distinct from—other connectives appearing in this work and others. The most obvious relative is $\Box$, which also encodes universal reasoning. In fact, $\Box A$ implies $\mathfrak{B}A$ and vice versa, in the proof theory. There are a number of differences, however. First, in the lambda calculus $\Box A$ does not imply $\mathfrak{B}A$, because of its restriction to values. Secondly, the fact that the bound world variable ω may appear in the type (because of the hybrid connective $A\,\mathtt{at}\,\mathrm{w}$) means that $\mathfrak{B}_\omega$ is more precise than $\Box$. This means that when the variable appears, we can't even state the equivalence $\mathfrak{B}_\omega A = \Box A$—let alone prove it—because A mentions a variable not in scope. The bound variable does not occur often in programs, but is important in the internals of the compiler.

Another question is whether the elimination form for $\Box$ ought to use the validity judgment like the elimination form for $\mathfrak{B}$. This is what Jia and Walker do [62] and it would be sound in our proof theory as well. However, the body of a `box` must be an *expression* for it to be the $\Box$ connective, and we do not want to have to substitute expressions for valid variables (Section 3.5.2). Therefore we prefer to use the formulation where each use of $\Box A$ is an explicit `unbox` that evaluates the expression when it is unboxed. This leaves the logical meaning of $\Box$ intact in our call-by-value language.

The shamrock modality was inspired by Park's $\boxdot$ connective [100, 101]. A value of type $\boxdot A$ is a suspended computation that, when evaluated, results in a value that is portable to any world.[3] The modality is defined in terms of a validity judgment, but unlike the validity judgment here, there are structured conclusions of validity. Validity can be concluded by (possibly effectful) expressions of primitive type (which corresponds to our mobile judgment and the `put` construct) or by a small language of pure expressions. This means that $\boxdot A$ itself would not be mobile in our setting, since the evaluation of its contents can have effects. The proof theory uses a form of leftist substitution where the substitution operation is defined inductively on the term being substituted, which does not appear to admit a straightforward compilation strategy. It is not known whether the logic containing $\boxdot$ is globally sound and complete (equivalent to a sequent calculus with the subformula property and cut). In comparison, we treat $\mathfrak{B}$ as an encapsulated *value* that is already portable to every world, and use a form of substitution that can be readily implemented using standard compiler techniques.

Finally, note that we cannot achieve the effect of validity by adding new types or by using polymorphism. For example, if we had quantification over worlds (which will be introduced in the next chapter) this would not be sufficient. Suppose we had a function f that we want to be able to use anywhere. We cannot simply bind it as a modal hypothesis $f{:}\forall\omega.\mathtt{int} \supset \mathtt{int}\,@\,\omega$. This judgment is ill-formed because the scope of the quantifier does not include $@\omega$, which is part of the *judgment*, not the *type*. Using the type $f{:}\forall\omega.(A\,\mathtt{at}\,\omega)$ also does not work, because the modal judgment requires us to place f at some specific world and then move it around explicitly. Therefore, we really need a new judgment.

[3] He formerly called this the $\bigcirc$ modality, which is why it appears that way in my Thesis Proposal [86] and other places.

3.6 Summary

In this chapter I have presented a series of logics and calculi that form the theoretical basis for the programming language ML5. For each proof theory I have demonstrated its logical status by proving it equivalent to a known logic, or to a natural sequent calculus with the subformula property. Each logic is accompanied by a lambda calculus derived from its proof terms. Although these lambda calculi are not always as strong (or direct) as the proof theory, we make such concessions with our eyes open, and justify the gap in terms of a dynamic semantics that suggests a straightforward path towards implementation on a distributed set of hosts. For each lambda calculus we provide a type safety result. Every proof has been formalized in a machine-checkable form.

We now shift gears slightly. In the next chapter, I investigate the process of compiling a small programming language based on the calculi in this chapter. I do so in the abstract, defining the compilation relations only for simple language features, without any regard to convenience, efficiency or other implementation concerns. Each of these compilation passes is performed in Twelf, with formalized type safety and type preservation results. However, we do not seek out connections with logic in this chapter. Once I have justified the major phases of compilation for this little language, I then present the full-scale programming language ML5 in Chapter 5, at which time I will be mainly concerned with pragmatic and not theoretical issues.

Chapter 4

Modal typed compilation

ML5 is implemented by compilation, which consists of a series of translations into simpler and simpler intermediate languages. The compiler is type-directed, meaning that these intermediate languages are themselves typed. In this chapter, I present a minimal programming language based on Lambda 5 called MinML5 and show how it can be compiled. This means defining a few intermediate languages, proving type safety for them, and defining type-preserving translations between the languages. Specifically, these translations are an elaboration phase to remove some derived forms from the external language (Section 4.4), CPS conversion (Section 4.5), and closure conversion (Section 4.7). Each language and translation is defined and shown to be (statically) correct in Twelf.

The language in this chapter is too minimal to write real programs in, and the compilation relations that I define are too inefficient to be used in a real compiler. Therefore, the actual implementation of ML5 (Chapter 5) is more complex, and I only give informal arguments for its correctness. However, this formal account of compilation is quite useful. For one, it is naturally part of the informal argument (by analogy) that the implementation is correct. Additionally, it exercises the language's expressiveness in a way that we have not done yet, because CPS and closure conversion make nontrivial use of the language features. We also find that as we compile the constructs of the language into simpler ones, those simpler ones are often more precise and thus more useful than the high-level constructs they encode. In fact, this exploration of compilation is what informed our addition of the `at` and $\mathfrak{B}_\omega$ modalities, as well as the `put` construct, into the external language.

Let's begin by seeing how the $\Box$ and $\Diamond$ connectives lack precision, and why the `at` modality is more expressive. We'll then present MinML5 and define the first translation, which expands $\Box$ and $\Diamond$ as derived forms for quantification and `at`.

4.1 The `at` modality

We included the `at` modality in Lambda 5 and justified it as a logical connective via the sequent calculus. However, we did not provide much motivation for it. There are sev-

eral good reasons to include it; for one, it is a natural result of using an explicit worlds formulation of modal logic, because it is a direct internalization of the @ judgment. This makes it not seem very exotic, since such internalizations are commonplace: For example, in propositional logic the $\supset$ connective is an internalization of the $\vdash$ judgment for hypothetical reasoning. Related to this point, we will see that the `at` modality is necessary for closure conversion in Section 4.7.

From a high-level programming point of view, `at` is useful because it is very precise. To illustrate, consider the following theorem of IS5:

$$\Box(A \supset B) \supset \Diamond A \supset \Diamond B$$

The reading of this proposition is as follows: Given that A implies B in every world, and given that A is true in some world, B is true in some world. This proposition is true by virtue of the fact that B is true at the same world that A is true (we go to that world and apply the mobile function from A to B). However, the proposition does not state that B is true at the *same* world; in fact, we unable to state such a thing with the $\Box$ and $\Diamond$ connectives at all. In this sense the $\Box$ and $\Diamond$ connectives, being unable to mention worlds, are "lossy."

The `at` modality allows us to make such statements. The MinML5 internal language introduces quantification over worlds, so we will be able to say

$$\Box(A \supset B) \supset \forall\omega.(A\,\texttt{at}\,\omega) \supset (B\,\texttt{at}\,\omega)$$

which is stronger than the above. In fact, in the logic the $\Box$ and $\Diamond$ modalities are definable in terms of `at` and quantification:

$$\Box A = \forall\omega.A\,\texttt{at}\,\omega$$
$$\Diamond A = \exists\omega.A\,\texttt{at}\,\omega$$

In the translation that follows, we will eliminate $\Box$ and $\Diamond$ by using `at` and quantification over worlds (the translation will be a bit more complicated than the above due to other operational concerns), and then not consider them again.

4.2 MinML5 external language

MinML5 is a miniature programming language based on Lambda 5. It has all of the connectives from Lambda 5 except for disjunction (which would not cause any special problems; I simply omit it for simplicity) plus the validity extensions. A new addition will be world addresses.

4.2.1 Addresses

The worlds in the typing judgment are the static representations of hosts in the network. At runtime, we will need tokens with which to refer to these worlds. A value of type

$$
\begin{array}{rrcl}
\text{variables} & x, y & & \\
\text{valid variables} & u & & \\
\text{valid values} & s & ::= & \omega.v \mid u \\
\text{expressions} & M, N & ::= & \texttt{here}\ M \mid M\ N \mid \texttt{leta}\ x = M\ \texttt{in}\ N \mid \texttt{lets}\ u = M\ \texttt{in}\ N \mid \\
 & & & \texttt{letd}\ \omega, x, y = M\ \texttt{in}\ N \mid \texttt{val}\ v \mid \texttt{unbox}\ M \mid \\
 & & & \texttt{put}\ u = M\ \texttt{in}\ N \mid \texttt{get}[M]\ N \mid \langle M, N \rangle \\
\text{values} & v & ::= & \boldsymbol{\lambda} x.M \mid \mathbf{held}\ v \mid \mathbf{sham}\ \omega.v \mid \mathbf{vval}\ s \mid x \\
 & & & \mathbf{there}[\overline{\mathrm{w}}, v] \mid \mathbf{box}\ \omega.M \mid \langle v_1, v_2 \rangle
\end{array}
$$

Figure 4.1: MinML5 external language syntax

$\mathrm{w}\ \texttt{addr}$ is such a token, which acts as the address of a host. It is a singleton type; there is only one value of type $\mathrm{w}\ \texttt{addr}$ for a given w. A host can compute its own address by using the $\texttt{localhost}()$ primitive, which results in an address value, written $\overline{\mathrm{w}}$:

$$\frac{}{\Gamma \vdash \texttt{localhost}() : \mathrm{w}\ \texttt{addr}\ @\mathrm{w}}\ \text{localhost} \qquad \frac{}{\Gamma \vdash \overline{\mathrm{w}} : \mathbf{w}\ \texttt{addr}\ @\mathrm{w}'}\ \text{address}$$

The way an address is used is by switching the location of evaluation to the world described by the address, using `get`. Thus, `get` now takes a new argument, which is the address of the remote world:

$$\frac{\begin{array}{l} A\ \mathsf{mobile} \\ \Gamma \vdash M : \mathrm{w}'\ \texttt{addr}@\mathrm{w} \\ \Gamma \vdash N : A@\mathrm{w}' \end{array}}{\Gamma \vdash \texttt{get}[M]N : A@\mathrm{w}}\ \text{get}$$

Addresses are themselves mobile. The `put` construct does not need an address, because it does not communicate with a remote host. However, the elimination form for $\Diamond$ now binds both the static world variable and its address; otherwise, we would be unable to contact that world in the body. The introduction form of $\Box$ does not need to be changed, because its body can use `localhost()` to find out where it is being evaluated, if desired.

4.2.2 Syntax and static semantics

The MinML5 external language syntax appears in Figure 4.1. The elements are familiar: We have syntactic classes of values and expressions, where some constructors have both expression and value forms (for example the pairing expression $\langle M, N \rangle$ and the paired value $\langle v_1, v_2 \rangle$). We have added addresses for worlds, as described above. As a minor technical difference, we now have a syntactic class s of valid value, which can be either a valid variable u or a value $\omega.v$ parametric in its world. We therefore have a judgment

$$\Gamma \vdash s \sim \omega.A$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathtt{val}\ v : A@\mathrm{w}}\ \text{val}$$

$$\frac{i \in \{1,2\} \quad \Gamma \vdash M : A_1 \wedge A_2@\mathrm{w}}{\Gamma \vdash \#i\ M : A_i@\mathrm{w}}\ \wedge \mathrm{E}_i \qquad \frac{\Gamma \vdash M_1 : A@\mathrm{w} \quad \Gamma \vdash M_2 : B@\mathrm{w}}{\Gamma \vdash \langle M_1, M_2 \rangle : A \wedge B@\mathrm{w}}\ \wedge \mathrm{I}$$

$$\frac{\Gamma \vdash M : A \supset B@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}}{\Gamma \vdash M\ N : B@\mathrm{w}}\ \supset\ \mathrm{E}$$

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{hold}\ M : A\ \mathtt{at}\ \mathrm{w}@\mathrm{w}}\ \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma \vdash M : A\ \mathtt{at}\ \mathrm{w}''@\mathrm{w}' \quad \Gamma, x{:}A@\mathrm{w}'' \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{leta}\ x = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \mathtt{at}\ \mathrm{E}$$

$$\frac{\Gamma \vdash M : \Box A@\mathrm{w}}{\Gamma \vdash \mathtt{unbox}\ M : A@\mathrm{w}}\ \Box\ \mathrm{E} \qquad \frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{here}\ M : \Diamond A@\mathrm{w}}\ \Diamond\ \mathrm{I}$$

$$\frac{\Gamma \vdash M : \Diamond A@\mathrm{w} \quad \Gamma, \omega\ \mathsf{world}, x{:}\omega\ \mathtt{addr}\ @\mathrm{w}, y{:}A@\omega \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{letd}\ \omega, x, y = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \Diamond\ \mathrm{E}$$

$$\frac{\Gamma \vdash M : \text{♁}_\omega A@\mathrm{w} \quad \Gamma, u \sim \omega.A \vdash N : C@\mathrm{w}}{\Gamma \vdash \mathtt{letsham}\ u = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \text{♁}\ \mathrm{E}$$

$$\frac{\begin{array}{l}A\ \mathsf{mobile}\\ \Gamma \vdash M : A@\mathrm{w}\\ \Gamma, u \sim A \vdash N : C@\mathrm{w}\end{array}}{\Gamma \vdash \mathtt{put}\ u = M\ \mathtt{in}\ N : C@\mathrm{w}}\ \text{put} \qquad \frac{\begin{array}{l}A\ \mathsf{mobile}\\ \Gamma \vdash M : \mathrm{w}'\ \mathtt{addr}\ @\mathrm{w}\\ \Gamma \vdash N : A@\mathrm{w}'\end{array}}{\Gamma \vdash \mathtt{get}[M]\ N : A@\mathrm{w}}\ \text{get}$$

$$\frac{}{\Gamma \vdash \mathtt{localhost}() : \mathrm{w}\ \mathtt{addr}\ @\mathrm{w}}\ \text{localhost}$$

Figure 4.2: The MinML external language typing rules for expressions

$$\frac{}{\Gamma, x{:}A@\mathrm{w} \vdash x : A@\mathrm{w}}\ \text{hyp} \qquad \frac{}{\Gamma, u \sim \omega.A \vdash u \sim \omega.A}\ \text{vhyp}$$

$$\frac{}{\Gamma \vdash \overline{\mathbf{w}} : \mathbf{w}\ \mathtt{addr}\ @\mathrm{w}'}\ \text{address} \qquad \frac{\Gamma \vdash s \sim \omega.A}{\Gamma \vdash \mathbf{vval}\ s : [{}^{\mathrm{w}}/_{\omega}]A@\mathrm{w}}\ \text{vval}$$

$$\frac{\Gamma \vdash v_1 : A@\mathrm{w} \quad \Gamma \vdash v_2 : B@\mathrm{w}}{\Gamma \vdash \langle v_1, v_2 \rangle : A \wedge B@\mathrm{w}} \wedge \mathrm{I}_v \qquad \frac{\Gamma, x{:}A@\mathrm{w} \vdash M : B@\mathrm{w}}{\Gamma \vdash \boldsymbol{\lambda} x.M : A \supset B@\mathrm{w}} \supset \mathrm{I}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}'}{\Gamma \vdash \mathbf{held}\ v : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{I}_v \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash M : A@\omega}{\Gamma \vdash \mathbf{box}\ \omega.M : \Box A@\mathrm{w}} \Box \mathrm{I}$$

$$\frac{\Gamma \vdash \overline{\mathbf{w}} : \mathbf{w}\ \mathtt{addr} \quad \Gamma \vdash v : A@\mathbf{w}}{\Gamma \vdash \mathbf{there}[\overline{\mathbf{w}}, v] : \Diamond A@\mathrm{w}} \Diamond \mathrm{I}_v \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash s : A@\omega}{\Gamma \vdash \mathbf{sham}\ \omega.s : \text{⟀}_{\omega} A@\mathrm{w}} \text{⟀}\, \mathrm{I}$$

Figure 4.3: The MinML external language typing rules for values and valid values

for the well-formedness of valid values. The rules for valid values and modal values appear in Figure 4.3. Valid values are values and values are expressions, via the inclusions `vval` and `val`. The typing rules for expressions appear in Figure 4.2.

4.2.3 Dynamic semantics

I do not give a dynamic semantics for the MinML5 external language, for two reasons. First, it would mostly be redundant to the semantics for Lambda 5 (except for the addition of addresses) and the MinML5 internal language. Simply note that in the semantics for Lambda 5, the evaluation rule for `get` only knew where to evaluate the subexpression by virtue of the world annotation on the `get` construct. This is somewhat unrealistic, since worlds are intended to be a purely static part of the type system. Now, because we have a runtime address indicating the destination, the dynamic semantics need not depend on static worlds at all.

The second reason is that there is an easy shortcut to defining an external language dynamic semantics, by composition of the elaboration relation with the evaluation relation of the internal language. In the next section we present the first internal language and its dynamic semantics, so that we can define the translation from the external language into it.

4.3 MinML5 internal language

The MinML5 internal language is similar to the external language, but some constructs have been added and some have been removed. Its syntax appears in Figure 4.4. First, we introduce type-level quantification over worlds. We have the types

$$\forall \omega.A \qquad \text{and} \qquad \exists \omega.A$$

$$
\begin{array}{rrcl}
\text{variables} & x, y & & \\
\text{valid variables} & u & & \\
\text{valid values} & s & ::= & \omega.v \mid u \\
\text{expressions} & M, N & ::= & M\ N \mid \texttt{leta}\ x = M\ \texttt{in}\ N \mid \texttt{lets}\ u = M\ \texttt{in}\ N \mid \\
 & & & \texttt{let}\ x = M\ \texttt{in}\ N \mid \texttt{val}\ v \mid M\langle \mathrm{w} \rangle \mid \texttt{unpack}\ \omega, x = M\ \texttt{in}\ N \mid \\
 & & & \texttt{put}\ u = M\ \texttt{in}\ N \mid \texttt{get}[M]\ N \\
\text{values} & v & ::= & \boldsymbol{\Lambda}\omega.M \mid \boldsymbol{\lambda} x.M \mid \mathbf{pack}\ \mathbf{w}, v\ \mathbf{as}\ \exists\omega.A \mid \mathbf{held}\ v \mid \\
 & & & \langle\rangle \mid \langle v_1, v_2 \rangle \mid \mathbf{sham}\ \omega.v \mid \mathbf{vval}\ s \mid x
\end{array}
$$

Figure 4.4: MinML5 internal language syntax

$$\frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \texttt{val}\ v : A@\mathrm{w}}\ \text{val} \qquad \frac{i \in \{1,2\} \quad \Gamma \vdash M : A_1 \wedge A_2@\mathrm{w}}{\Gamma \vdash \#i\ M : A_i@\mathrm{w}}\ \wedge \mathrm{E}_i$$

$$\frac{\Gamma \vdash M : A \supset B@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}}{\Gamma \vdash M\ N : B@\mathrm{w}}\ \supset \mathrm{E} \qquad \frac{\Gamma \vdash M : A@\mathrm{w} \quad \Gamma, x{:}A@\mathrm{w} \vdash N : C@\mathrm{w}}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : C@\mathrm{w}}\ \text{let}$$

$$\frac{\Gamma \vdash M : \forall\omega.A@\mathrm{w}}{\Gamma \vdash M\langle \mathrm{w}' \rangle : [{}^{\mathrm{w}'}/_{\omega}]A@\mathrm{w}}\ \forall \mathrm{E}$$

$$\frac{\Gamma \vdash M : \exists\omega.A@\mathrm{w} \quad \Gamma, \omega\ \mathsf{world}, x{:}A@\mathrm{w} \vdash N : C@\mathrm{w} \quad (\omega \notin \mathrm{FV}(C))}{\Gamma \vdash \texttt{unpack}\ \omega, x = M\ \texttt{in}\ N : C@\mathrm{w}}\ \exists \mathrm{E}$$

$$\frac{\Gamma \vdash M : A\ \texttt{at}\ \mathrm{w}''@\mathrm{w}' \quad \Gamma, x{:}A@\mathrm{w}'' \vdash N : C@\mathrm{w}}{\Gamma \vdash \texttt{leta}\ x = M\ \texttt{in}\ N : C@\mathrm{w}}\ \texttt{at}\ \mathrm{E}$$

$$\frac{\Gamma \vdash M : \text{ƹ}_{\omega}A@\mathrm{w} \quad \Gamma, u \sim \omega.A \vdash N : C@\mathrm{w}}{\Gamma \vdash \texttt{letsham}\ u = M\ \texttt{in}\ N : C@\mathrm{w}}\ \text{ƹ}\ \mathrm{E}$$

$$\frac{\begin{array}{l}A\ \mathsf{mobile} \\ \Gamma \vdash M : A@\mathrm{w} \\ \Gamma, u \sim A \vdash N : C@\mathrm{w}\end{array}}{\Gamma \vdash \texttt{put}\ u = M\ \texttt{in}\ N : C@\mathrm{w}}\ \text{put} \qquad \frac{\begin{array}{l}A\ \mathsf{mobile} \\ \Gamma \vdash M : \mathrm{w}'\ \texttt{addr}\ @\mathrm{w} \\ \Gamma \vdash N : A@\mathrm{w}'\end{array}}{\Gamma \vdash \texttt{get}[M]\ N : A@\mathrm{w}}\ \text{get}$$

$$\frac{}{\Gamma \vdash \texttt{localhost}() : \mathrm{w}\ \texttt{addr}\ @\mathrm{w}}\ \text{localhost}$$

Figure 4.5: The MinML internal language typing rules for expressions

$$\frac{}{\Gamma, x{:}A@\mathrm{w} \vdash x : A@\mathrm{w}}\ \text{hyp} \qquad \frac{\Gamma \vdash s \sim \omega.A}{\Gamma \vdash \mathbf{vval}\ s : [\mathrm{w}/\omega]A@\mathrm{w}}\ \text{vval}$$

$$\frac{\Gamma \vdash v_1 : A@\mathrm{w} \quad \Gamma \vdash v_2 : B@\mathrm{w}}{\Gamma \vdash \langle v_1, v_2 \rangle : A \wedge B@\mathrm{w}}\ \wedge\,\mathrm{I} \qquad \frac{\Gamma, x{:}A@\mathrm{w} \vdash M : B@\mathrm{w}}{\Gamma \vdash \boldsymbol{\lambda} x.M : A \supset B@\mathrm{w}}\ \supset\,\mathrm{I}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}'}{\Gamma \vdash \mathbf{held}\ v : A\,\mathtt{at}\,\mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\mathrm{w}}{\Gamma \vdash \boldsymbol{\Lambda}\omega.v : \forall\omega.A@\mathrm{w}}\ \forall\,\mathrm{I}$$

$$\frac{}{\Gamma \vdash \overline{\mathbf{w}} : \mathbf{w}\ \mathtt{addr}\ @\mathrm{w}'}\ \text{address} \qquad \frac{\Gamma \vdash v : [\mathbf{w}/\omega]A@\mathrm{w}}{\Gamma \vdash \mathbf{pack}\ \mathbf{w}, v\ \mathbf{as}\ \exists\omega.A : \exists\omega.A@\mathrm{w}}\ \exists\,\mathrm{I}$$

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}@\mathrm{w}}\ \mathtt{unit}\,\mathrm{I} \qquad \frac{\Gamma, \vdash s \sim \omega.A}{\Gamma \vdash \mathbf{sham}\ s : \mathfrak{B}_\omega A@\mathrm{w}}\ \mathfrak{B}\,\mathrm{I}$$

$$\frac{}{\Gamma, u{\sim}\omega.A \vdash u \sim \omega.A}\ \text{vhyp} \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\omega}{\Gamma \vdash \omega.v \sim \omega.A}\ \text{valid}$$

Figure 4.6: The MinML internal language typing rules for values and valid values.

$$\frac{A\ \mathsf{mobile} \quad B\ \mathsf{mobile}}{A \wedge B\ \mathsf{mobile}}\ \wedge\,\mathrm{M} \qquad \frac{A\ \mathsf{mobile}}{\forall\omega.A\ \mathsf{mobile}}\ \forall\,\mathrm{M} \qquad \frac{A\ \mathsf{mobile}}{\exists\omega.A\ \mathsf{mobile}}\ \exists\,\mathrm{M}$$

$$\frac{}{\mathfrak{B}_\omega A\ \mathsf{mobile}}\ \mathfrak{B}_\omega\mathrm{M} \qquad \frac{}{\mathrm{w}\ \mathtt{addr}\ \mathsf{mobile}}\ \mathtt{addr}\ \mathrm{M} \qquad \frac{}{A\,\mathtt{at}\,\mathrm{w}\ \mathsf{mobile}}\ \mathtt{at}\ \mathrm{M}$$

$$\frac{}{\mathtt{unit}\ \mathsf{mobile}}\ \mathtt{unit}\ \mathrm{M}$$

Figure 4.7: The definition of the mobile judgment for the MinML5 internal language.

for universal and existential quantification, respectively, where ω is bound within A. These quantified types are mobile only if the body is mobile (Figure 4.7). We also add a standard `let` construct and the `unit` type.

Some features in the external language will be translated away, which means that they do not appear in the internal language. The $\Box$ and $\Diamond$ connectives do not appear; they will be translated as below. We also have no need for expressions that construct pairs or values of `at` type; we can always sequence the evaluation of the body with `let` and then construct the value. This leaves us with a nice system where all of the introduction forms are values and all of the elimination forms are expressions.

The type system appears in Figures 4.5 and 4.6. The rules for world quantification are straightforward. To type check $\boldsymbol{\Lambda}$, we simply assume the hypothetical world and check the body. To check `pack`, we verify that it has the abstract type from the annotation if we substitute in the actual world. Note that in both cases the body is required to be a value; this is because we do not want these constructs to have any run-time behavior. To use a term of type $\boldsymbol{\Lambda}\omega.A$, we can apply it to a static world expression, resulting in that instance

of A. To use a term of type $\exists\omega.A$, we bind the hypothetical world ω, and a value of type A. Note that this `unpack` construct does not bind an address for the abstract world, so our translation from `letd` will need to do that manually.

4.3.1 Dynamic semantics

For the MinML5 internal language I give a small-step dynamic semantics based on evaluation contexts. This semantics is much simpler than the one for C5 (Section 3.4.6) because we do not use tables to store values, instead simply using substitution. The syntax and typing rules for evaluation contexts appear in Figure 4.8. The judgment

$$\Gamma \vdash e : A \star \mathrm{w}$$

states that the evaluation context e is well-formed and expects a value of type A in the world w. A machine m is then

$$e \,|_{\mathbf{w}}\, M$$

and is well-formed (in the empty context) if its evaluation context and expression agree:

$$\frac{\cdot \vdash e : A \star \mathbf{w} \quad \cdot \vdash M : A @ \mathbf{w}}{\cdot \vdash e \,|_{\mathbf{w}}\, M \;\mathsf{ok}} \text{ machine}$$

The evaluation relation $\mapsto_{\mathbf{w}}$ is a relation between two machines, indexed by the world at which evaluation takes place. It is defined in Figure 4.9.

Rules roughly fall into two categories. The first is where the machine state consists of an evaluation context and unevaluated expression, in which case we extend the evaluation context and begin evaluating the expression. The second is where the machine state consists of an evaluation context and a value appropriate for that context (via the `val` inclusion), in which case we perform the reduction and continue. For `localhost()`, we extract the world from the machine state (this assumes that every world can compute its own address, which is reasonable) and return it. We use this same ability in the rule for `get` to supply the `return` frame with the address of the world it should return to.

The `get` and `put` use a partial function lift to make a value of mobile type into a valid value (so these rules only apply when lift is defined). The definition of lift appears in Figure 4.10. It is a straightforward induction that adds an unused world argument, except in the case where the value is actually a **vval**, in which case we instantiate it at the current world and then lift that value instead.

The final rule deserves some attention. If we have a valid value being used as a value (via the inclusion **vval**), then we can instantiate it with the current world to get a regular value. We are only forced to use this rule when we otherwise require a specific form of value (for example in the reduction rule for `leta`); otherwise, we can delay its application. This causes minor trouble in the proof of safety because it means that we have multiple canonical forms for a given type: for $A \supset B$ we have $\boldsymbol{\lambda} x.M$ and **vval** $\omega.v$ where v is a canonical form for $A \supset B$. (On the other hand, we do not have

evaluation contexts

$$\begin{array}{rcl} e & ::= & \texttt{finish} \mid e;\, \texttt{get}[\circ]\ M \mid e;\, \texttt{return}[\mathbf{w}]\ \circ \mid e;\, \texttt{put}\ u = \circ\ \texttt{in}\ N \mid \\ & & e;\, \circ\ N \mid e;\, v\ \circ \mid e;\, \circ\langle \mathrm{w} \rangle \mid e;\, \#1\ \circ \mid e;\, \#2\ \circ \mid \\ & & e;\, \texttt{leta}\ x = \circ\ \texttt{in}\ N \mid e;\, \texttt{let}\ x = \circ\ \texttt{in}\ N \mid \\ & & e;\, \texttt{lets}\ u = \circ\ \texttt{in}\ N \mid e;\, \texttt{unpack}\ \omega, x = \circ\ \texttt{in}\ N \end{array}$$

$$\frac{}{\Gamma \vdash \texttt{finish} : A \star \mathrm{w}}\ \text{finish}$$

$$\frac{A\ \mathsf{mobile} \quad \Gamma \vdash e : A \star \mathrm{w} \quad \Gamma \vdash M : A @ \mathrm{w}'}{\Gamma \vdash e;\, \texttt{get}[\circ]\ M : \mathrm{w}'\ \texttt{addr} \star \mathrm{w}}\ \text{get}_1 \qquad \frac{A\ \mathsf{mobile} \quad \Gamma \vdash e : A \star \mathbf{w}}{\Gamma \vdash e;\, \texttt{return}[\mathbf{w}] \circ : A \star \mathrm{w}}\ \text{get}_2$$

$$\frac{A\ \mathsf{mobile} \quad \Gamma \vdash e : C \star \mathrm{w} \quad \Gamma, u{\sim}A \vdash N : C @ \mathrm{w}}{\Gamma \vdash e;\, \texttt{put}\ u = \circ\ \texttt{in}\ N : A \star \mathrm{w}}\ \text{put}$$

$$\frac{\Gamma \vdash e : B \star \mathrm{w} \quad \Gamma \vdash N : A @ \mathrm{w}}{\Gamma \vdash e;\, \circ\ N : A \supset B \star \mathrm{w}}\ \text{app}_1 \qquad \frac{\Gamma \vdash e : B \star \mathrm{w} \quad \Gamma \vdash v : A \supset B @ \mathrm{w}}{\Gamma \vdash e;\, v \circ : A \star \mathrm{w}}\ \text{app}_2$$

$$\frac{\Gamma \vdash e : [{}^{\mathrm{w}'}/_{\omega}]A \star \mathrm{w}}{\Gamma \vdash e;\, \circ\langle \mathrm{w}' \rangle : \forall \omega.A \star \mathrm{w}}\ \text{wapp}$$

$$\frac{\Gamma \vdash e : A \star \mathrm{w}}{\Gamma \vdash e;\, \#1 \circ : A \wedge B \star \mathrm{w}}\ \#1 \qquad \frac{\Gamma \vdash e : B \star \mathrm{w}}{\Gamma \vdash e;\, \#2 \circ : A \wedge B \star \mathrm{w}}\ \#2$$

$$\frac{\Gamma \vdash e : C \star \mathrm{w} \quad \Gamma, x{:}A @ \mathrm{w}' \vdash N : C @ \mathrm{w}}{\Gamma \vdash e;\, \texttt{leta}\ x = \circ\ \texttt{in}\ N : A\ \texttt{at}\ \mathrm{w}' \star \mathrm{w}}\ \text{leta}$$

$$\frac{\Gamma \vdash e : C \star \mathrm{w} \quad \Gamma, x{:}A @ \mathrm{w} \vdash N : C @ \mathrm{w}}{\Gamma \vdash e;\, \texttt{let}\ x = \circ\ \texttt{in}\ N : A \star \mathrm{w}}\ \text{let}$$

$$\frac{\Gamma \vdash e : C \star \mathrm{w} \quad \Gamma, u{\sim}\omega.A \vdash N : C @ \mathrm{w}}{\Gamma \vdash e;\, \texttt{lets}\ u = \circ\ \texttt{in}\ N : \mathfrak{B}_{\omega} A \star \mathrm{w}}\ \text{lets}$$

$$\frac{\Gamma \vdash e : C \star \mathrm{w} \quad \Gamma, \omega\ \mathsf{world}, x{:}A @ \mathrm{w} \vdash N : C @ \mathrm{w}}{\Gamma \vdash e;\, \texttt{unpack}\ \omega, x = \circ\ \texttt{in}\ N : \exists \omega.A \star \mathrm{w}}\ \text{unpack}$$

Figure 4.8: Syntax and typing for evaluation contexts. The type A in the judgment $\Gamma \vdash e : A \star \mathrm{w}$ is the type that the context *expects* (the type of the "hole" $\circ$).

$$
\begin{array}{rcl}
e;\ \texttt{unpack}\,\omega, x = \circ\ \texttt{in}\ N\,|_{\mathbf{w}} \\
\quad \texttt{val}\ (\mathbf{pack}\ \mathbf{w}', v\ \mathbf{as}\ \exists\omega.A) & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,[{}^{\mathbf{w}'}\!/_{\omega}][{}^{v}\!/_{x}]N \\
e\,|_{\mathbf{w}}\,\texttt{unpack}\,\omega, x = M\ \texttt{in}\ N & \mapsto_{\mathbf{w}} & e;\ \texttt{unpack}\,\omega, x = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,M \\
e\,|_{\mathbf{w}}\,\texttt{localhost}() & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,\texttt{val}\ \overline{\mathbf{w}} \\
e\,|_{\mathbf{w}}\,\#1\ M & \mapsto_{\mathbf{w}} & e;\ \#1\ \circ\ |_{\mathbf{w}}\,M \\
e\,|_{\mathbf{w}}\,\#2\ M & \mapsto_{\mathbf{w}} & e;\ \#2\ \circ\ |_{\mathbf{w}}\,M \\
e;\ \#1\ \circ\ |_{\mathbf{w}}\,\texttt{val}\ \langle v_1, v_2\rangle & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,v_1 \\
e;\ \#2\ \circ\ |_{\mathbf{w}}\,\texttt{val}\ \langle v_1, v_2\rangle & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,v_2 \\
e\,|_{\mathbf{w}}\,M\ N & \mapsto_{\mathbf{w}} & e;\ \circ\ N\,|_{\mathbf{w}}\,M \\
e;\ \circ\ N\,|_{\mathbf{w}}\,\texttt{val}\ v & \mapsto_{\mathbf{w}} & e;\ v\ \circ\ |_{\mathbf{w}}\,N \\
e;\ (\boldsymbol{\lambda}x.M)\ \circ\ |_{\mathbf{w}}\,\texttt{val}v & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,[{}^{v}\!/_{x}]M \\
e\,|_{\mathbf{w}}\,M\langle \mathrm{w}\rangle & \mapsto_{\mathbf{w}} & e;\ \circ\langle \mathrm{w}\rangle\,|_{\mathbf{w}}\,M \\
e;\ \circ\langle \mathrm{w}\rangle\,|_{\mathbf{w}}\,\texttt{val}\ \boldsymbol{\Lambda}\omega.v & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,\texttt{val}[{}^{\mathrm{w}}\!/_{\omega}]v \\
e\,|_{\mathbf{w}}\,\texttt{put}\ u = M\ \texttt{in}\ N & \mapsto_{\mathbf{w}} & e;\ \texttt{put}\ u = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,M \\
e;\ \texttt{put}\ u = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,\texttt{val}\ v & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,[{}^{\mathsf{lift}\ \mathbf{w}\ v}\!/_{u}]N \\
e\,|_{\mathbf{w}}\,\texttt{get}[M]\ N & \mapsto_{\mathbf{w}} & e;\ \texttt{get}[\circ]\ N\,|_{\mathbf{w}}\,M \\
e;\ \texttt{get}[\circ]\ N\,|_{\mathbf{w}}\,\texttt{val}\ \overline{\mathbf{w}'} & \mapsto_{\mathbf{w}} & e;\ \texttt{return}[\mathbf{w}]\ \circ\ |_{\mathbf{w}'}\,N \\
e;\ \texttt{return}[\mathbf{w}]\ \circ\ |_{\mathbf{w}'}\,\texttt{val}\ v & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,\texttt{val}\ (\mathbf{vval}(\mathsf{lift}\ \mathbf{w}'\ v)) \\
e\,|_{\mathbf{w}}\,\texttt{let}\ x = M\ \texttt{in}\ N & \mapsto_{\mathbf{w}} & e;\ \texttt{let}\ x = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,M \\
e;\ \texttt{let}\ x = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,\texttt{val}\ v & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,[{}^{v}\!/_{x}]N \\
e\,|_{\mathbf{w}}\,\texttt{leta}\ x = M\ \texttt{in}\ N & \mapsto_{\mathbf{w}} & e;\ \texttt{leta}\ x = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,M \\
e;\ \texttt{leta}\ x = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,\texttt{val}\ \mathbf{held}\ v & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,[{}^{v}\!/_{x}]N \\
e\,|_{\mathbf{w}}\,\texttt{lets}\ u = M\ \texttt{in}\ N & \mapsto_{\mathbf{w}} & e;\ \texttt{lets}\ u = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,M \\
e;\ \texttt{lets}\ u = \circ\ \texttt{in}\ N\,|_{\mathbf{w}}\,\texttt{val}\ \mathbf{sham}\ s & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,[{}^{s}\!/_{u}]N \\
e\,|_{\mathbf{w}}\,\texttt{val}\ (\mathbf{vval}\ \omega.v) & \mapsto_{\mathbf{w}} & e\,|_{\mathbf{w}}\,\texttt{val}\ [{}^{\mathbf{w}}\!/_{\omega}]v
\end{array}
$$

Figure 4.9: The small-step evaluation relation for MinML5 internal language machines.

$$
\begin{array}{lcl}
\mathsf{lift}\ \mathbf{w}\ (\mathbf{held}\ v) & = & \omega.(\mathbf{held}\ v) \\
\mathsf{lift}\ \mathbf{w}\ \langle v_1, v_2\rangle & = & \omega.\langle \mathbf{vval}\ (\mathsf{lift}\ \mathbf{w}\ v_1), \mathbf{vval}\ (\mathsf{lift}\ \mathbf{w}\ v_2)\rangle \\
\mathsf{lift}\ \mathbf{w}\ (\boldsymbol{\Lambda}\omega'.v) & = & \omega.\boldsymbol{\Lambda}\omega'.(\mathbf{vval}\ (\mathsf{lift}\ \mathbf{w}\ v_1)) \\
\mathsf{lift}\ \mathbf{w}\ (\mathbf{pack}\ \mathrm{w}, v\ \mathbf{as}\ \exists\omega'.A) & = & \omega.(\mathbf{pack}\ \mathrm{w}, (\mathbf{vval}\ (\mathsf{lift}\ \mathbf{w}\ v))\ \mathbf{as}\ \exists\omega'.A) \\
\mathsf{lift}\ \mathbf{w}\ \overline{\mathbf{w}'} & = & \omega.\overline{\mathbf{w}'} \\
\mathsf{lift}\ \mathbf{w}\ (\mathbf{sham}\ s) & = & \omega.\mathbf{sham}\ s \\
\mathsf{lift}\ \mathbf{w}\ (\mathbf{vval}\ \omega'.v) & = & \mathsf{lift}\ \mathbf{w}\ ([{}^{\mathbf{w}}\!/_{\omega'}]v)
\end{array}
$$

Figure 4.10: The definition of the partial function lift, which takes a world and a value, and returns a valid value.

to define a special validity substitution operation as in Section 3.5.) The reason that this nondeterminism is acceptable is that we have in mind an implementation where valid values and values have the same representation, meaning that this instantiation has no run-time effect.

The MinML5 internal language is type safe:

Theorem 33 (Type preservation of $\mapsto_{\mathbf{w}}$)

If $\mathcal{D} :: \cdot \vdash e\,|_{\mathbf{w}}\, M$ ok
and $\mathcal{E} :: e\,|_{\mathbf{w}}\, M \mapsto_{\mathbf{w}} e'\,|_{\mathbf{w}'}\, M'$
then $\mathcal{F} :: \cdot \vdash e'\,|_{\mathbf{w}'}\, M'$ ok

Theorem 34 (Progress for $\mapsto_{\mathbf{w}}$)

If $\mathcal{D} :: \cdot \vdash e\,|_{\mathbf{w}}\, M$ ok
then either $\mathcal{E} :: e\,|_{\mathbf{w}}\, M \mapsto_{\mathbf{w}} e'\,|_{\mathbf{w}'}\, M'$ for some e', $\mathbf{w}'$, and M
or $e\,|_{\mathbf{w}}\, M$ is terminal

The only terminal state is $\texttt{finish}\,|_{\mathbf{w}}\,\texttt{val}\ v$. Proof of Theorem 33 is an easy induction on $\mathcal{E}$ and appears as the Twelf relation `presv` in Appendix A.8.3. The only thing to mention is that we need a lemma that the lift operation produces a well-typed valid value if its input is well-typed:

Theorem 35 (Type preservation of lift)

If $\mathcal{D} :: \Gamma \vdash v : A@\mathbf{w}$
and $\mathcal{E} :: \mathsf{lift}\ \mathbf{w}\ v = s$
then $\mathcal{F} :: \Gamma \vdash s : \omega.A$

Proof is straightforward by induction on $\mathcal{E}$, and the full proof appears as the Twelf relation `momo` in Appendix A.8.3.

Proof of Theorem 34 is also straightforward. The necessary lemma is that $\mathsf{lift}\ \mathbf{w}\ v$ is defined whenever v has mobile type:

Theorem 36 (Totality of lift on values of mobile type)

If $\mathcal{D} :: \cdot \vdash v : A@\mathbf{w}$
and $\mathcal{E} :: A$ mobile
then $\mathcal{F} :: \mathsf{lift}\ \mathbf{w}\ v = s$ *(*for some s*)*

It is also an easy induction, appearing as the relation `mobmov` in Appendix A.8.3. To conveniently state the progress theorem, we use the standard trick of defining the evaluation relation in Twelf to relate a terminal machine to itself. The totality of the relation encodes the disjunctive theorem statement, by either encoding $\mapsto_{\mathbf{w}}$ when one of its steps applies, or by self-stepping when the machine is terminal. The proof appears as the Twelf relation `presv` in Appendix A.8.3. □

4.4 Elaboration

Having defined the external and internal languages, we can now give the elaboration relation. The translation is type-directed, so it is only defined for well-typed terms, and is defined inductively on the external language typing derivation rather than the

$$
\begin{array}{rcl}
[\![A \wedge B]\!]_{\mathsf{E}} & = & [\![A]\!]_{\mathsf{E}} \wedge [\![B]\!]_{\mathsf{E}} \\
& \vdots & \\
[\![\Box A]\!]_{\mathsf{E}} & = & \forall\omega.((\mathtt{unit} \to [\![A]\!]_{\mathsf{E}})\,\mathtt{at}\,\omega) \\
[\![\Diamond A]\!]_{\mathsf{E}} & = & \exists\omega.([\![A]\!]_{\mathsf{E}}\,\mathtt{at}\,\omega \wedge \omega\,\mathtt{addr})
\end{array}
$$

Figure 4.11: The elaboration relation from external language types to internal language types.

term. Since we lack a convenient notation for functions on typing derivations, I will give the translation informally on the syntax and assume access to some parts of the typing derivation (types and worlds) that are necessary. The formal relation is defined in Twelf, and is discussed briefly in Section 4.4.1.

We define a collection of relations, relating external language types, expressions, values, and valid values to their internal language counterparts. Since most of the external language constructs have a directly analogous counterpart in the internal language, I avoid giving those cases except for illustration.

The translation of types is a good starting point, since it shows how $\Box$ and $\Diamond$ will be treated. It is given as a function $[\![\cdot]\!]_{\mathsf{E}}$, defined for all well-formed external language types, and appears in Figure 4.11. Most cases are pointwise, as the case given for $\wedge$. A value of type $\Box A$ is a suspended computation that can be evaluated at any world. Therefore we translate this type to a function taking the trivial `unit` argument (a suspension) which is encapsulated with `at` to mark that it makes sense at some other world, and then that world is made polymorphic with $\forall$. A value of type $\Diamond A$ is a value that makes sense at some abstract world. We pair that value with a dynamic address for that world, and then pack it into the existential type to hide the world. (Recall that the external language constructs for $\Box$ do not do anything with addresses, since the suspended expression can just compute its address with `localhost()` if desired.) Because $\Box A$ and $\Diamond A$ are mobile for any A, it is important that their translations be mobile as well. This is the case here because $A\,\mathtt{at}\,\mathrm{w}$ is also mobile for any A.

Next is the elaboration relation for values, given as a function $[\![\cdot]\!]_{\mathsf{VE}}^{A@\mathrm{w}}$ defined for all well-typed external language values, and indexed by the world and type of the value. It appears along with the elaboration relation for expressions and valid values (with which it is mutually recursive) in Figure 4.12. The translation of the **box** and **there** constructors is straightforward knowing the translation of the types. The only thing we must be careful about is that a translated value is also a value, because we continue to have syntactic value restrictions in the intermediate language (particularly on **held** and $\mathbf{\Lambda}$). The translation of valid values is given by $[\![\cdot]\!]_{\mathsf{SE}}^{\omega:A}$ and has only two cases, both of which are pointwise.

The translation of expressions is given by $[\![\cdot]\!]_{\mathsf{E}}^{A@\mathrm{w}}$ and also indexed by the world and type of the expression. Note how for the expression constructors $\langle M, N\rangle$ and `hold` we just sequence the evaluation of the body and then use the value constructors, obviating the need for these constructors in the internal language.

For `unbox`, we apply the translated value (which is polymorphic in its world) to

$$
\begin{array}{rcl}
[\![\mathbf{box}\omega.M]\!]^{\Box A@\mathrm{w}}_{\mathsf{VE}} & = & \mathbf{\Lambda}\omega.(\mathbf{held}\ \lambda x.[\![M]\!]^{A@\omega}_{\mathsf{E}}) \\
[\![\mathbf{there}[\overline{\mathrm{w}}, v]]\!]^{\Diamond A@\mathrm{w}}_{\mathsf{VE}} & = & \mathbf{pack}\ \mathrm{w}, \langle \mathbf{held}\ [\![v]\!]^{A@\mathrm{w}}_{\mathsf{VE}}, \overline{\mathrm{w}} \rangle \\
 & & \quad \mathbf{as}\ \exists\omega.([\![A]\!]_{\mathsf{E}}\ \mathtt{at}\ \omega \wedge \omega\ \mathtt{addr}) \\
 & \vdots & \\
\\
[\![\langle M, N\rangle]\!]^{A\wedge B@\mathrm{w}}_{\mathsf{E}} & = & \mathtt{let}\ x = [\![M]\!]^{A@\mathrm{w}}_{\mathsf{E}}\ \mathtt{in} \\
 & & \mathtt{let}\ y = [\![N]\!]^{B@\mathrm{w}}_{\mathsf{E}}\ \mathtt{in} \\
 & & \quad \langle x, y\rangle \\
[\![\mathtt{hold}\ M]\!]^{A\,\mathtt{at}\,\mathrm{w}@\mathrm{w}}_{\mathsf{E}} & = & \mathtt{let}\ x = [\![M]\!]^{A@\mathrm{w}}_{\mathsf{E}}\ \mathtt{in}\ \mathbf{held}\ x \\
[\![\mathtt{unbox}\ M]\!]^{A@\mathrm{w}}_{\mathsf{E}} & = & \mathtt{leta}\ x = [\![M]\!]^{\Box A@\mathrm{w}}_{\mathsf{E}}\langle \mathrm{w}\rangle\ \mathtt{in}\ x\ \langle\rangle \\
[\![\mathtt{here}\ M]\!]^{\Diamond A@\mathrm{w}}_{\mathsf{E}} & = & \mathtt{let}\ y = [\![M]\!]^{A@\mathrm{w}}_{\mathsf{E}}\ \mathtt{in} \\
 & & \mathtt{let}\ x = \mathtt{localhost}()\ \mathtt{in} \\
 & & \quad \mathbf{pack}\ \mathrm{w}, \langle \mathbf{held}\ y, x\rangle \\
 & & \quad \mathbf{as}\ \exists\omega.([\![A]\!]_{\mathsf{E}}\ \mathtt{at}\ \omega \wedge \omega\ \mathtt{addr}) \\
[\![\mathtt{letd}\ \omega, x, y = M\ \mathtt{in}\ N]\!]^{C@\mathrm{w}}_{\mathsf{E}} & = & \mathtt{unpack}\ \omega, x' = [\![M]\!]^{\Diamond A@\mathrm{w}}_{\mathsf{E}}\ \mathtt{in} \\
 & & \quad \mathtt{let}\ x = \#2\ x'\ \mathtt{in} \\
 & & \quad \mathtt{leta}\ y = \#1\ x'\ \mathtt{in}\ [\![N]\!]^{C@\mathrm{w}}_{\mathsf{E}} \\
 & \vdots & \\
\\
[\![u]\!]^{\omega.A}_{\mathsf{SE}} & = & u \\
[\![\omega.v]\!]^{\omega.A}_{\mathsf{SE}} & = & \omega.[\![v]\!]^{A@\omega}_{\mathsf{VE}}
\end{array}
$$

Figure 4.12: The elaboration relation from the external language to internal language, for values, expressions, and valid values.

the current world. This gives us an expression of type $(\texttt{unit} \supset A)\,\texttt{at}\,\mathrm{w}$, which we consume by using `leta`. Note that because this term is "`at`" the current world, when we bind it with `leta` we get a local hypothesis which we can use immediately. We then apply that function to $\langle\rangle$, which evaluates the suspension. The translation of `here` is like `there`, but we sequence the evaluation of the expression and use `localhost`() to compute the current world's address before packing the existential. The elimination form, `letd`, simply unpacks the term, then projects out the stored address and value. Observe that in this case our inductive call to $[\![M]\!]^{\Diamond A@\mathrm{w}}_{\mathsf{E}}$ must apparently "guess" the type A. This is why (for example) the translation is actually defined on type derivations; in the derivation that the `letd` expression is well-typed, we can simply read off the type $\Diamond A$ from the subderivation that M is well-typed.

Elaboration is type-correct and defined for all well-formed types, values, expressions, and valid values:

Theorem 37 (Functionality of type elaboration)
$[\![A]\!]_{\mathsf{E}}$ is defined and unique for all well-formed A

Theorem 38 (Static correctness of expression elaboration)
If $\mathcal{D} :: \Gamma \vdash M : A@\mathrm{w}$ *(*in the external language*)*
then $[\![v]\!]^{A@\mathrm{w}}_{\mathsf{VE}} = M'$ *(*for some M'*)*
and $[\![\Gamma]\!] \vdash M' : [\![A]\!]_{\mathsf{E}}@\mathrm{w}$ *(*in the internal language*)*

Theorem 39 (Static correctness of value elaboration)
If $\mathcal{D} :: \Gamma \vdash v : A@\mathrm{w}$ *(*in the external language*)*
then $[\![v]\!]^{A@\mathrm{w}}_{\mathsf{VE}} = v'$ *(*for some v'*)*
and $[\![\Gamma]\!] \vdash v' : [\![A]\!]_{\mathsf{E}}@\mathrm{w}$ *(*in the internal language*)*

Theorem 40 (Static correctness of valid value elaboration)
If $\mathcal{D} :: \Gamma \vdash s \sim \omega.A$ *(*in the external language*)*
then $[\![s]\!]^{\omega.A}_{\mathsf{SE}} = s'$ *(*for some s'*)*
and $[\![\Gamma]\!] \vdash s' \sim \omega.[\![A]\!]_{\mathsf{E}}$ *(*in the internal language*)*

The operation $[\![\Gamma]\!]$ simply translates the types of the variables in Γ in the obvious way. The informal proof of each theorem is by induction on the typing derivation (or for Theorem 37, the structure of the type). The formal proof comes from the totality of the Twelf translation relations on typing derivations, which is discussed in the next section.

4.4.1 Elaboration in Twelf

Although all of the theorems in this dissertation so far have been proved in Twelf, this is the first Twelf formalization that I will discuss in any detail. This particular proof is fairly easy, but its simplicity makes it an appropriate way to set the stage for the next two translations, which are much less simple. The two important lessons here are the (somewhat nonstandard) methodology of using the computational content of a proof to *define* a translation, and a particular trick for defining type-directed translations. Both

lessons will be important for understanding the Twelf encodings of CPS (Section 4.6.2) and closure conversion (Section 4.7.1).

Brief introduction to Twelf

To refresh the reader's memory, I give a brief and informal tour of Twelf and how it can be used to formalize programming languages and their metatheory. For a more complete and formal discussion, I recommend Harper and Licata's *Mechanizing Metatheory in a Logical Framework* [52].

Twelf is an implementation of the LF logical framework [51], which allows us to encode judgments as type families in a dependently-typed lambda calculus. We can encode a programming language's syntax as a type family, and then encode the language's typing judgment in terms of its syntax [105]. For example, a fragment of the MinML5 internal language is as follows:

```
world : type.
ty    : type.
val   : type.

unit  : ty.
at    : ty -> world -> ty.

1     : val.
held  : val -> val.
```

We define three type families, for the syntactic classes of worlds, types, and values (`type` is a Twelf keyword, whereas `ty` is the name of the syntactic class in MinML5). We then declare that the constant `unit` is a `ty`, and that the constant `at`, applied to a type and a world, is a `ty`. Finally, we declare the value `1` and value constructor `held`. Having made these definitions, we can then define the typing judgment using the same mechanisms:

```
ofv : val -> ty -> world -> type.

unitI : ofv 1 unit W.
atI   : ofv V A W' ->
        ofv (held V) (A at W') W.
```

The type family `ofv` is a three place relation between a value, a type, and a world, with the intention of encoding the typing judgment $\Gamma \vdash v : A @ \mathrm{w}$. If we want to prove that the `ofv` type family really encodes the typing judgment, we can prove a bijection between canonical LF terms and typing derivations; this theorem is known as *adequacy*. I do not give any adequacy theorems in this dissertation because I consider the LF formalization to be primary and inference rules to be merely presentational.

The constants that inhabit `ofv` correspond to the inference rules; `unitI` is the rule concluding that $\langle\rangle$ (here written `1`) has type `unit`. We can conclude this in any world; the capital existential variable `W` is syntax for an implicit argument to this constructor

indicating the world. The `atI` constructor is similar, but takes a *subderivation* that the embedded value is well-typed.

Because Twelf is also a logic programming system, we can define relations on terms that have computational content. For example, supposing we have declarations for external language values, well-typedness (suffixed with the letter `e`; for example, `vale`, and `ofve`). We can then define an elaboration relation for values:

```
elabv : vale -> val -> type.
%mode elabv +VE -VI.

elabv/1 : elabv 1e 1.
elabv/held : elabv (helde V) (held V')
          <- elabv V V'.
```

This type family relates an external language value with an internal language value. By declaring its mode, we assert that the relation can be run as a program, where the external language value is an input (+) and the internal language value is an output (−). Running this relation as a program means requesting a derivation of `elabv v VI` where `v` is a canonical LF term of type `vale` and `VI` is an existential variable. Twelf's operational semantics defines how to search the constants defining `elabv` to find a matching derivation. In the case that the input is `(helde 1e)`, for example, search sees that only `elabv/held` could be the outermost constructor on the derivation. Since this constructor needs an argument, it recursively searches for a derivation of `elabv 1 V'`, finds that `elabv/1` matches, and so returns the derivation `elabv/held elabv/1`.

Metatheorems

To prove a metatheorem about a relation, we use Twelf's facilities for verifying totality assertions [52, 121]. For instance, we typically want to know that a translation is type preserving. We can state this for `elabv` as an LF relation:

```
elabok : ofve V A W ->
         elabv V V' ->
         ofv  V' A W ->
         type.
%mode elabok +WV +EL -WV'.
```

The idea is that `elabok` relates three derivations: the derivation that `V` is well formed, the derivation that `V` translates to `V'`, and a derivation that `V'` is well-formed at the same type and world. What we want to express is that *if* there are derivations for first two positions (the inputs) *then* there is a derivation for the output position. We use the relation's mode to indicate the inputs and outputs, as above. The proof of this theorem has two parts. We first give constants defining this relation, such that Twelf's search strategy will always find a derivation of `elabok` it when a query is made with canonical input derivations. We then use Twelf's metatheory system to verify that our proof (a program that transforms the input derivations into the output derivation) always succeeds. This property is called totality.

The cases for this proof (constants inhabiting `elabok`) are:

```
elabok/1    : elabok unitIe elabv/1 unitI.
elabok/held : elabok (atIe WV) (elabv/held E) (atI WV')
           <- elabok WV E WV'.
```

To verify that this relation is total, we first specify what sorts of LF terms may appear in the context:

```
%block w : block {W : world}.
%worlds (w) (elabok WV EL WV').
```

In Twelf terminology, the "world" is a description of the context. This is somewhat confusing in the current work, where we use "world" in a completely different way. I therefore say "context" except when using the `%worlds` keyword.

In this case we assume only hypotheses of the existence of object language worlds (without this, there would be no derivations of `ofv` at all, since we declared no constants of type `world`). The `%worlds` declaration checks that the relation recursively maintains this form of context. We can then assert the totality of the relation `elabok`.

```
%total EL (elabok WV EL WV').
```

The parameter `EL` is the induction metric, which in this case is just the structure of the second input. Internally, Twelf checks several properties when verifying a totality assertion. First, it checks that for any query matching the mode, some case of the proof matches it. It then checks that any subgoals (premises of the case) obey the induction metric, so that proof search always makes progress. (There are also some subtle restrictions on the form of subgoals, which I do not discuss here.) If the totality assertion succeeds, then we know formally that for all canonical LF terms in the input positions (living in the context we declared), there exists a canonical LF term in the output position. If we believe that the LF terms adequately represent what we intend, then we have a proof of an "If... then" statement. In this case, we have proved that if $\Gamma \vdash v : A@\mathrm{w}$ and $[\![v]\!] = v'$ then $\Gamma \vdash v' : A@\mathrm{w}$, provided that Γ takes the form ω_i world $\ldots \omega_n$ world.

Exploiting the computational content of "metatheorems"

The totality assertion is typically used to prove metatheorems in the manner described above [50, 52]. In fact, the meaning of the totality assertion is somewhat stronger, which gives us a shortcut to developing some kinds of metatheory. The important fact is that the totality assertion does not just mean "for all inputs, there exist outputs," but also that there is a derivation of the inductively-proved theorem that specifically relates those inputs to the outputs. What this means is that it matters which proof we give for a particular theorem, because it describes *how* we get the outputs from the inputs. I exploit this computational content to make the translation and its proof of static correctness the same entity.

Forgetting the definitions `elabv` and `elabok`, let's define the following relation between external and internal language typing derivations:

```
delabv : ofve V A W -> ofv V' A W -> type.
%mode delabv +WV -WV'
```

```
delabv/1 : delabv unitIe unitI.
delabv/held : delabv (atIe WV) (atI WV')
           <- delabv WV WV'.

%worlds () (delabv _ _).
%total D (delabv D _).
```

Taken as a metatheorem of the above form, the totality of this relation is uninteresting: it states that if there is some well-typed term in the external language, then there is some well-typed term in the internal language. This statement is particularly weak if we have a construct like `raise` in our language, where we can create an abortive term at any type! However, if we consider that the totality assertion means that *this* relation is total, then we get as much as we did in the previous development. Because it translates the typing derivations it also translates the terms within them, and because the Twelf relation is well-typed, the resultant terms are manifestly well-typed. Because the relation is total, the translation succeeds for *every* well-typed term.

I use this technique as often as I can, because it means writing fewer relations, the content of which would be somewhat redundant. (For instance, observe that `elabok` could treat the translation relation as an *output* argument, or leave it out entirely!) There are a few ways in which this approach might be considered inferior, however. First, we do not give any translation to ill-typed terms. This is not a problem, since we expect to only try to compile well-typed terms. (In comparison, the previous formulation did not give any translation for syntactically ill-formed terms, which is considered normal.) Second, if we wish for the translation to only depend on the term (because we want a coherence property or because when we implement it we expect to not have the typing derivation around), this formulation makes it harder to see that. This is also not an issue, for two reasons: Either way we have to prove a coherence or functionality result if we want it, and in the specific domain of type-directed compilers, by their very nature we do generally depend on more than just the term.

4.4.2 The elaboration relations

Let us now look at the real elaboration relations. Figure 4.13 summarizes the Twelf type families that encode the syntax and judgments of the external and internal languages. Given these, we can define the type translation relation:

```
ettoit : tye -> ty -> type.
%mode ettoit +TE -TI.
```

This translation just relates an external language type to its translated internal language type and is defined in the obvious way. We need it to define the translations for expression, value, and valid value typing derivations. The translation for values is, for example,

```
elabv+ : {A:tye} {V : vale} ofve V A W ->
         ettoit A A' ->
```

Language	Declaration	Meaning
EL IL	`world : type`	Worlds
EL IL	`tye : type` `ty : type`	Types
EL IL	`vale : type` `val : type`	Values
EL IL	`expe : type` `exp : type`	Expressions
EL IL	`vvale : type` `vval : type`	Valid values
EL IL	`ofe : expe -> tye -> world -> type` `of : exp -> ty -> world -> type`	Typing of exps
EL IL	`ofve : vale -> tye -> world -> type` `ofv : val -> ty -> world -> type`	Typing of values
EL IL	`ofvve : vvale -> (world -> tye) -> type` `ofvv : vval -> (world -> ty) -> type`	Typing of valid values

Figure 4.13: Twelf type families defining the external language and internal language.

```
          {VV : val} ofv VV A' W -> type.
%mode elabv+ +A +B +BW +BT -E -OE.
elabv- : {A:tye} {V : vale} ofve V A W ->
         ettoit A A' ->
         {VV : val} ofv VV A' W -> type.
%mode elabv- +A +B +BW -BT -E -OE.
```

The two relations `elabv+` and `elabv-` have the same type, but different modes—the type translation from `A` to `A'` is an input in the first and an output in the second. Ultimately we only care about the first one, but we need both in order to prove the theorem. This mutually inductive definition is the second technique that I want to point out, and it is used in several of the proofs.

The reason that we do this is subtle, and ultimately comes down to the fact that the relation `ettoit` is a function but that Twelf does not know this unless we prove it. We can begin by stating that lemma because we use it later:

```
ettoit_fun : ettoit A A' -> ettoit A A'' ->
             eqtyp A' A'' -> type.
%mode ettoit_fun +X +Y -Z.
```

Given two translations of `A`, this theorem supplies a proof that they produce equal results. It is an easy induction over the two input derivations. However, we must prove this lemma; though we wrote the translation on paper using suggestive notation, it is not automatic that it is a function. (For example, it might have overlapping cases that produce different results.) Additionally, we need to know that `ettoit` can always successfully be run on a type:

```
ettoit_gimme : {A:tye} {A':ty} ettoit A A' -> type.
%mode ettoit_gimme +A -A' -D.
```

This is known as an "effectiveness" lemma [68] and is also easy to prove.

A failed attempt

To see why we must have both an input and output version of our theorems, let us try to prove `elab+` and `elabv+` directly. We will consider the cases for application and lambdas. Application does work:

```
- : elab+ _ _ (=>Ee Df Da) ETb _ (=>E Df' Da')
 <- ettoit_gimme A A' ETa
 <- elab+ _ _ Df (ettoit_=> ETa ETb) _ Df'
 <- elab+ _ _ Da ETa _ Da'.
```

The application of a function of type `A => B` has type `B`, so we get an input derivation `ETb : ettoit B B'`. To elaborate the function and argument, we also need a derivation of `ETa : ettoit A A'`. An appeal to the functionality lemma gives us this, and then we can apply induction on both components.

The lambda case comes close to working, but fails because of the way variables are handled in Twelf. A (very) naïve attempt is as follows

```
- : elabv+ _ _ (=>Ie D) (ettoit_=> ETa ETb) _ (=>I D')
 <- ({ve : vale}{ove : ofve ve A W}
     {v  : val} {ov  : ofv  v  A' W}
       elab+ _ _ (D ve ove) ETb _ (D' v ov)).
```

Since we are translating a lambda, we know that the external language type is of the form `A => B`, and therefore its translation takes the form `ettoit_=> ETa ETb` (`ettoit_=>` is the case of the translation for arrow types, and takes derivations of the translations of `A` and `B`). This gives us a translation derivation for `B` which we can pass off to translate the body. However, in doing so we have introduced EL variables into the context, which means that we need to say how to elaborate them. This is accomplished by adding an elaboration derivation into the context along with the variable and then declaring (with the `%worlds` declaration) that EL variables are always accompanied by elaboration derivations. The standard thing to try here would be:

```
- : elabv+ _ _ (=>Ie D) (ettoit_=> ETa ETb) _ (=>I D')
 <- ({ve : vale}{ove : ofve ve A  W}
     {v  : val} {ov  : ofv  v  A' W}
     {thm : elabv+ A ve ove ETa v ov}
       elab+ _ _ (D ve ove) ETb _ (D' v ov)).

%block blockve :
  some  {A:tye}{A':ty}{W:world}{ETv : ettoit A A'}
  block {ve : vale}{ove : ofve ve A  W}
        {v  : val} {ov  : ofv  v  A' W}
        {thm : elabv+ A ve ove ETv v ov}.
```

```
%worlds (blockve) (elab+ _ _ _ _ _ _) (elabv+ _ _ _ _ _ _).
```

This does not work. The reason is somewhat subtle. To check that our proof covers all of the cases, Twelf looks at the constants defining our language and the description of the context given by the `%worlds` declaration to see what possible forms the input might take. It then checks to see that the input is matched either by one of the cases of the proof, or by a case that must be in the context because of its regular form. For `elabv+`, the inputs are the type, value typing derivation, and type translation. This includes variables from the block declaration we just made, so we must be able to match

```
elabv+ A ve ove ET _ _
```

for some arbitrary `A : tye`, `A' : tye` and `ET : ettoit A A'`. None of the cases will match variables because they all case-analyze the form of the derivation. Therefore, only the case that we inserted for the variables could match. Unfortunately, it doesn't. The derivation `ET` of `ettoit A A'` that we're trying to match isn't necessarily the same as the one that we put into the context with the variables; in fact, it doesn't necessarily even translate `A` to the same `A'`. Input coverage therefore fails. Although the reasoning to show that this doesn't matter (because `ettoit` is a function) is easy to carry out, we have no place to do it, so this attempt is a dead end.

Reversing the polarity

Instead we will carry out the cases for `elab-` and `elabv-`. Here the lambda case works out cleanly:

```
- : elabv- _ _ (=>Ie D) (ettoit_=> ETa ETb) _ (=>I D')
 <- ettoit_gimme A A' ETa
 <- ({ve : vale}{ove : ofve ve A W}
     {v  : val} {ov  : ofv  v  A' W}
     {thm : elabv- A ve ove ETa v ov}
       elab- _ _ (D ve ove) ETb _ (D' v ov)).

%block blockve :
  some  {A:tye}{A':ty}{W:world}{ETv : ettoit A A'}
  block {ve : vale}{ove : ofve ve A  W}
        {v  : val} {ov  : ofv  v  A' W}
        {thm : elabv- A ve ove ETv v ov}.
```

Because we must output the type translation, we sometimes must appeal to effectiveness, as we do here. When we descend under the binder, we introduce external language and internal language variables, and the case for elaboration that relates them. Since the type translation derivation is now an *output*, it is not considered during input coverage checking and therefore does not suffer from the problem above.

In essence, we have changed the meaning of the theorem from "give me any type translation you like and I will elaborate using it" to "I will elaborate and tell you which

type translation I used." Since there is only one translation of any type, these two are actually the same, but it is easier to check coverage for the second. We carry out the equality reasoning to implement the `elab+` relations in terms of the `elab-` ones. This is a single case that covers all inputs:

```
of_resp : of M A W -> eqtyp A A' -> of M A' W -> type.
%mode of_resp +BOF +EQ -BOF'.

el : elab+ A M WM BTi E OE'
  <- elab- A M WM BTo E OE
  <- ettoit_fun BTo BTi EQ
  <- of_resp OE EQ OE'.
```

The `of_resp` lemma states that typing derivations respect the equality of types, and is trivial to prove. To implement `elab+`, we receive the type translation as an input (`BTi`). We immediately invoke `elab-` on the same inputs, getting the elaborated expression but a second type translation `BTo`. We use functionality to prove that they translate `A` to equal types. We then use the fact that typing derivations respect equality to produce a derivation that agrees with the input type translation, and return that.

Having these two versions of the lemma is convenient when writing the cases for the `elab-` family as well, because it saves us from having to carry out equality reasoning. For example, the case for application is as follows:

```
- : elab- _ _ (=>Ee Df Da) ETb _ (=>E Df' Da')
 <- elab- _ _ Df (ettoit_=> ETa ETb) _ Df'
 <- elab+ _ _ Da ETb _ Da'.
```

We must produce as output `ETb : ettoit B B'`, the translation of the result type. We start by recursing on the typing derivation for the function, using the "`-`" version of elaboration. This gives us a translation for the function type `A => B`, which contains as subderivations the translations of `A` and `B`. Since we already have a translation of `B`, we use `elab+` to translate the function's argument. If we appealed to `elabv-` again, we would then need to reason that the translation for `B` that it produced the same result as the translation we already have. This is easy to do, but it is nicer to use the two different versions as appropriate.

Theorem

The rest of the cases go through easily using these same techniques. (The elaboration of valid values only needs a "`-`" version because is syntax is so simple.) Once they are complete, we can verify the totality all of the (mutually inductive) elaboration relations at once:

```
%total (D1 D2 D3 D4 D5)  (elab- _ _ D1 _ _ _)
  (elab+ _ _ D2 _ _ _)   (elabv- _ _ D3 _ _ _)
  (elabv+ _ _ D4 _ _ _)  (elabvv _ _ D5 _ _ _).
```

Induction is on the typing derivation, as expected. The order of the relations here is important, because the one case of `elab+` appeals to `elab-` on a typing derivation

of the same size; therefore we must consider the `elab-` relation to be smaller in our simultaneous induction metric. The full proof appears in Appendix A.8.5. □

Elaboration is a simple translation, but its formalization displays some of the techniques that are important for the following translations, which are less straightforward. The first is conversion to continuation passing style, which begins in the next section.

4.5 Continuation passing style

The next phase of compilation is translation to continuation passing style (CPS), after which the remainder of the compiler operates on CPS internal languages. In a CPS language, the control stack is represented explicitly as a heap object. For this reason CPS-based compilers are somewhat rare—most production compilers are based on direct-style internal languages so that they can make use of the machine stack. For ML5, the CPS representation gives us numerous advantages. Because the control stack can span worlds, we cannot easily make use of the machine stack. It gives us an easy way to implement first-class continuations, which we use in some applications (Chapter 6). Exceptions are also easily translated away at this step via a double-barreled CPS translation (Section 5.4.2), which avoids us having to treat them in the remainder of the compiler. In the particular domain of web programming, CPS allows us to avoid restrictions on the stack depth and the length of computations in JavaScript, and allows us to implement threads in a straightforward way. For similar reasons, several related web languages such as Links [21] have CPS-based compilers.

The syntax of the CPS language appears in Figure 4.14. The language sequences the evaluation of expressions: Continuation expressions c mostly take the form of a `let` that performs a primitive operation on some number of values, binding some variables within the nested continuation expression. The continuation expression can end with a `halt` (ending the program) or a `call` to a continuation value on an argument. Calls do not return; the essence of the CPS translation is that the return continuation becomes part of the argument to the call. Additionally, the IL `get` construct (which evaluates an expression at a remote world and returns its value) has been replaced with the CPS language `go`. This construct transfers control (only) to a continuation at a remote world, and does not return. Values v are the same as in the IL, except for $\boldsymbol{\lambda} x.c$, which now encapsulates a continuation expression and so does not return when invoked. Such values have type A `cont`, which replaces the $A \supset B$ function type. It is similarly not mobile (Figure 4.15).

The well-formedness of valid values and values follows closely the typing of the internal language analogues; only the $\boldsymbol{\lambda} x.s$ construct differs (Figure 4.16). The typing rules for continuation expressions appear in Figure 4.17. The judgment

$$\Gamma \vdash c \star \mathrm{w}$$

states that the continuation c is well-formed for evaluation at w. These are mostly straightforward. To call a continuation (Rule `call`) we simply need an argument value

$$
\begin{array}{rrcl}
\text{conts} & c & ::= & \texttt{leta}\ x = v\ \texttt{in}\ c \\
& & \mid & \texttt{lets}\ u = v\ \texttt{in}\ c \\
& & \mid & \texttt{put}\ u = v\ \texttt{in}\ c \\
& & \mid & \texttt{let}\ x = \texttt{fst}\ v\ \texttt{in}\ c \\
& & \mid & \texttt{let}\ x = \texttt{snd}\ v\ \texttt{in}\ c \\
& & \mid & \texttt{let}\ x = \texttt{localhost}()\ \texttt{in}\ c \\
& & \mid & \texttt{let}\ x = v\langle \mathrm{w} \rangle\ \texttt{in}\ c \\
& & \mid & \texttt{let}\ \omega, x = \texttt{unpack}\ v\ \texttt{in}\ c \\
& & \mid & \texttt{go}[\mathrm{w}; v_a]\ c \\
& & \mid & \texttt{call}\ v_f(v_x) \\
& & \mid & \texttt{halt} \\
\text{values} & v & ::= & \langle v_1, v_2 \rangle \mid \langle\rangle \mid \mathbf{held}\ v \mid \mathbf{pack}\ \mathbf{w}, v\ \mathbf{as}\ \exists \omega.A \\
& & \mid & \boldsymbol{\lambda} x.c \mid \boldsymbol{\Lambda} \omega.v \mid \mathbf{vval}\ v \mid \overline{\mathbf{w}} \mid \mathbf{sham}\ s \mid x \\
\text{valid values} & s & ::= & u \mid \omega.v \\
\text{types} & A, B & ::= & A\ \texttt{cont} \mid A \wedge B \mid A\ \texttt{at}\ \mathrm{w} \mid \mathfrak{B}_\omega A \\
& & \mid & \forall \omega.A \mid \exists \omega.A \mid \mathrm{w}\ \texttt{addr} \mid \texttt{unit}
\end{array}
$$

Figure 4.14: The syntax of the CPS language.

$$
\frac{A\ \mathsf{mobile} \quad B\ \mathsf{mobile}}{A \wedge B\ \mathsf{mobile}}\ \wedge \mathrm{M}
\qquad
\frac{A\ \mathsf{mobile}}{\forall \omega.A\ \mathsf{mobile}}\ \forall\ \mathrm{M}
\qquad
\frac{A\ \mathsf{mobile}}{\exists \omega.A\ \mathsf{mobile}}\ \exists\ \mathrm{M}
$$

$$
\frac{}{\mathfrak{B}_\omega A\ \mathsf{mobile}}\ \mathfrak{B}_\omega \mathrm{M}
\qquad
\frac{}{\mathrm{w}\ \texttt{addr}\ \mathsf{mobile}}\ \texttt{addr}\ \mathrm{M}
\qquad
\frac{}{A\ \texttt{at}\ \mathrm{w}\ \mathsf{mobile}}\ \texttt{at}\ \mathrm{M}
$$

$$
\frac{}{\texttt{unit}\ \mathsf{mobile}}\ \texttt{unit}\ \mathrm{M}
$$

Figure 4.15: The definition of the mobile judgment for the CPS language.

$$\frac{}{\Gamma, x{:}A@\mathrm{w} \vdash x : A@\mathrm{w}}\ \text{hyp} \qquad \frac{\Gamma \vdash s \sim \omega.A}{\Gamma \vdash \mathbf{vval}\ s : [{}^{\mathrm{w}}/_{\omega}]A@\mathrm{w}}\ \text{vval}$$

$$\frac{\Gamma \vdash v_1 : A@\mathrm{w} \quad \Gamma \vdash v_2 : B@\mathrm{w}}{\Gamma \vdash \langle v_1, v_2 \rangle : A \wedge B@\mathrm{w}}\ \wedge \mathrm{I} \qquad \frac{\Gamma, x{:}A@\mathrm{w} \vdash c\star\mathrm{w}}{\Gamma \vdash \boldsymbol{\lambda} x.c : A\ \mathtt{cont}@\mathrm{w}}\ \supset\ \mathrm{I}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}'}{\Gamma \vdash \mathbf{held}\ v : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}\ \mathtt{at}\ \mathrm{I} \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\mathrm{w}}{\Gamma \vdash \boldsymbol{\Lambda}\omega.v : \forall\omega.A@\mathrm{w}}\ \forall\,\mathrm{I}$$

$$\frac{}{\Gamma \vdash \overline{\mathbf{w}} : \mathbf{w}\ \mathtt{addr}\ @\mathrm{w}'}\ \text{address} \qquad \frac{\Gamma \vdash v : [{}^{\mathbf{w}}/_{\omega}]A@\mathrm{w}}{\Gamma \vdash \mathbf{pack}\ \mathbf{w}, v\ \mathbf{as}\ \exists\omega.A : \exists\omega.A@\mathrm{w}}\ \exists\,\mathrm{I}$$

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}@\mathrm{w}}\ \mathtt{unit}\ \mathrm{I} \qquad \frac{\Gamma, \vdash s \sim \omega.A}{\Gamma \vdash \mathbf{sham}\ \omega.s : \text{ß}_{\omega}A@\mathrm{w}}\ \text{ß}\,\mathrm{I}$$

$$\frac{}{\Gamma, u{\sim}\omega.A \vdash u \sim \omega.A}\ \text{vhyp} \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\omega}{\Gamma \vdash \omega.v \sim \omega.A}\ \text{valid}$$

Figure 4.16: The CPS language typing rules for values and valid values.

of the right type; both must be at the current world. The continuation `halt` is well-typed at any world. Like `get`, the `go` construct requires the address of the target world (Rule `go`), and the subcontinuation must be well-typed there. However, it does not require its argument to be mobile—in fact, it has no type to insist is mobile in the first place! The `go` construct is only concerned with control flow, as will be clear from the dynamic semantics in the next section. The translation from `get` to `go` will encode the data transfer using `put`, which still requires its argument to be mobile. This translation is described in Section 4.6, immediately following the dynamic semantics.

4.5.1 Dynamic semantics

Again, we will give a dynamic semantics for the CPS internal language and prove that the language's typing rules induce type safety. Owing to the fact that the language has become simpler (as would be expected from a lower-level intermediate language), the semantics is easy to give. We do not need to introduce evaluation frames or abstract machines, simply a single-step evaluation relation between continuation expressions:

$$c \mapsto_{\mathbf{w}} c'$$

It is indexed by the concrete world at which evaluation is taking place, as usual. Because a value may be an uninstantiated valid value, we also need an evaluation relation for values.

$$v \rightsquigarrow_{\mathbf{w}} v'$$

This relation ensures that v' is not of the form $\mathbf{vval}\ s$, by instantiating s at the current world if v is of that form. Because we consider such instantiations to have no runtime

$$\frac{\Gamma \vdash v : A\,\mathtt{at}\,\mathrm{w}'@\mathrm{w} \quad \Gamma, x{:}A@\mathrm{w}' \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{leta}\,x = v\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{leta}$$

$$\frac{\Gamma \vdash v : \mathfrak{B}_\omega A@\mathrm{w} \quad \Gamma, u{\sim}\omega.A \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{lets}\,u = v\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{lets}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w} \quad A\ \mathsf{mobile} \quad \Gamma, u{\sim}A \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{put}\,u = v\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{put}$$

$$\frac{\Gamma \vdash v : A \wedge B@\mathrm{w} \quad \Gamma, x{:}A@\mathrm{w} \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{let}\,x = \mathtt{fst}\,v\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{fst}$$

$$\frac{\Gamma \vdash v : A \wedge B@\mathrm{w} \quad \Gamma, x{:}B@\mathrm{w} \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{let}\,x = \mathtt{snd}\,v\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{snd}$$

$$\frac{\Gamma, x{:}\mathrm{w}\,\mathsf{addr}\,@\mathrm{w} \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{let}\,x = \mathtt{localhost}()\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{lh}$$

$$\frac{\Gamma \vdash v : \forall\omega.A@\mathrm{w} \quad \Gamma, x{:}[{}^{\mathrm{w}'}/_{\omega}]A@\mathrm{w} \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{let}\,x = v\langle\mathrm{w}'\rangle\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{wapp}$$

$$\frac{\Gamma \vdash v : \exists\omega.A@\mathrm{w} \quad \Gamma, \omega\ \mathsf{world}, x{:}A@\mathrm{w} \vdash c\star\mathrm{w}}{\Gamma \vdash \mathtt{let}\,\omega, x = \mathtt{unpack}\,v\,\mathtt{in}\,c\star\mathrm{w}}\ \mathtt{unpack}$$

$$\frac{\Gamma \vdash v_a : \mathrm{w}'\,\mathsf{addr}\,@\mathrm{w} \quad \Gamma \vdash c\star\mathrm{w}'}{\Gamma \vdash \mathtt{go}[\mathrm{w}'; v_a]\,c\star\mathrm{w}}\ \mathtt{go}$$

$$\frac{\Gamma \vdash v_f : A\,\mathtt{cont}@\mathrm{w} \quad \Gamma \vdash v_a : A@\mathrm{w}}{\Gamma \vdash \mathtt{call}\,v_f(v_x)\star\mathrm{w}}\ \mathtt{call}$$

$$\frac{}{\Gamma \vdash \mathtt{halt}\star\mathrm{w}}\ \mathtt{halt}$$

Figure 4.17: The CPS language typing rules for continuation expressions.

$$\begin{array}{rcll} \mathbf{vval}\,\omega.v & \rightsquigarrow_{\mathbf{w}} & [{}^{\mathbf{w}}/_{\omega}]v & \\ v & \rightsquigarrow_{\mathbf{w}} & v & \text{iff } v \neq \mathbf{vval}\,\omega.v' \end{array}$$

Figure 4.18: The value instantiation relation for the CPS language. The world index on the relation is used when instantiating a valid value; however, this use has no run-time effect. I use a side condition to cover the cases for all value forms that are not **vval**, since these are the same.

significance, these steps are just coercions. The relation is defined in Figure 4.18.

The evaluation relation for continuation expressions appears in Figure 4.19. It uses a partial function lift (Rule `put`), which is defined identically to the one from the internal language in Figure 4.10 (so I do not repeat it here). Recall that lift hoists a value to a valid value, and that it is not defined for all inputs.

The evaluation relation also uses the value stepping relation we just defined. For elimination rules, we expect the input value to have a particular form; we achieve this form by appealing to the value stepping relation. For `go`, we simply check that the address matches the indicated world, and then start evaluating the nested continuation (the next step will therefore take place at the remote world).

4.5.2 Type safety

Type safety is straightforward for the CPS language.

Theorem 41 (Type preservation of $\mapsto_{\mathbf{w}}$)
If $\mathcal{D} :: \cdot \vdash c \star \mathbf{w}$
and $\mathcal{E} :: c \mapsto_{\mathbf{w}} c'$
then $\mathcal{F} :: \cdot \vdash c' \star \mathbf{w}'$ *(for some $\mathbf{w}'$)*

Theorem 42 (Progress for $\mapsto_{\mathbf{w}}$)
If $\mathcal{D} :: \cdot \vdash c \star \mathbf{w}$
then either $\mathcal{E} :: c \mapsto_{\mathbf{w}} c'$ *(for some c')*
or c is `halt`

For preservation (Theorem 41) we state that the output continuation is typed at *some* world, possibly different from the input (the `go` rule, particularly, changes the world). (If we were to state the theorem as a relation between abstract worlds, as before, then this world annotation would be the only additional data in the abstract machine.) For progress (Theorem 42) we know that a single step can be carried out at the same world at which the continuation is typed, or else that the continuation is `halt`.

These theorems need progress and preservation lemmas for the $\rightsquigarrow_{\mathbf{w}}$ relation, similar to the canonical forms properties in standard lambda calculi. We also require a lemma about lift, stated and proved analogously to Theorem 36. The full proofs of these lemmas and the type safety theorems appear in Appendix A.8.4. □

Having established that the CPS language is sound, we are now concerned with showing that it is expressive enough to encode the internal language. This is the process of CPS conversion, described in the next section.

$$\frac{v_f \rightsquigarrow_{\mathbf{w}} \boldsymbol{\lambda} x.c}{\texttt{call}\ v_f(v_a) \mapsto_{\mathbf{w}} [{}^{v_a}\!/_{x}]c}\ \texttt{call}$$

$$\frac{\mathsf{lift}\ \mathbf{w}v = v'}{\texttt{put}\ u = v\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{\omega.v'}\!/_{u}]c}\ \texttt{put}$$

$$\frac{v \rightsquigarrow_{\mathbf{w}} \mathbf{held}\ v'}{\texttt{leta}\ x = v\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{v'}\!/_{x}]c}\ \texttt{leta}$$

$$\frac{v \rightsquigarrow_{\mathbf{w}} \mathbf{sham}\ s}{\texttt{lets}\ u = v\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{s}\!/_{u}]c}\ \texttt{lets}$$

$$\frac{v \rightsquigarrow_{\mathbf{w}} \langle v_1, v_2 \rangle}{\texttt{let}\ x = \texttt{fst}\ v\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{v_1}\!/_{x}]c}\ \texttt{fst}$$

$$\frac{v \rightsquigarrow_{\mathbf{w}} \langle v_1, v_2 \rangle}{\texttt{let}\ x = \texttt{snd}\ v\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{v_2}\!/_{x}]c}\ \texttt{snd}$$

$$\frac{}{\texttt{let}\ x = \texttt{localhost}()\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{\overline{\mathbf{w}}}\!/_{x}]c}\ \texttt{lh}$$

$$\frac{v \rightsquigarrow_{\mathbf{w}} \boldsymbol{\Lambda}\omega.v'}{\texttt{let}\ x = v\langle \mathrm{w}' \rangle\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{[\mathrm{w}'/\omega]v'}\!/_{x}]c}\ \texttt{wapp}$$

$$\frac{v \rightsquigarrow_{\mathbf{w}} \mathbf{pack}\ \mathbf{w}', v'\ \mathbf{as}\ \exists\omega.A}{\texttt{let}\ \omega, x = \texttt{unpack}\ v\ \texttt{in}\ c \mapsto_{\mathbf{w}} [{}^{\mathbf{w}'}\!/_{\omega}][{}^{v'}\!/_{x}]c}\ \texttt{unpack}$$

$$\frac{v_a \rightsquigarrow_{\mathbf{w}} \overline{\mathbf{w}'}}{\mathsf{go}[\mathbf{w}'; v_a]\ c \mapsto_{\mathbf{w}} c}\ \mathsf{go}$$

Figure 4.19: The evaluation relation for CPS expressions.

$$
\begin{array}{rcl}
[\![A \supset B]\!]_{\mathsf{c}} & = & ([\![A]\!]_{\mathsf{c}} \wedge [\![B]\!]_{\mathsf{c}}\ \mathtt{cont})\ \mathtt{cont} \\
[\![A\ \mathtt{at}\ \mathrm{w}]\!]_{\mathsf{c}} & = & [\![A]\!]_{\mathsf{c}}\ \mathtt{at}\ \mathrm{w} \\
[\![\forall\omega.A]\!]_{\mathsf{c}} & = & \forall\omega.[\![A]\!]_{\mathsf{c}} \\
[\![\exists\omega.A]\!]_{\mathsf{c}} & = & \exists\omega.[\![A]\!]_{\mathsf{c}} \\
[\![\mathtt{unit}]\!]_{\mathsf{c}} & = & \mathtt{unit} \\
[\![A \wedge B]\!]_{\mathsf{c}} & = & [\![A]\!]_{\mathsf{c}} \wedge [\![B]\!]_{\mathsf{c}} \\
[\![\mathfrak{B}_\omega A]\!]_{\mathsf{c}} & = & \mathfrak{B}_\omega [\![A]\!]_{\mathsf{c}} \\
[\![\mathrm{w}\ \mathtt{addr}]\!]_{\mathsf{c}} & = & \mathrm{w}\ \mathtt{addr}
\end{array}
$$

Figure 4.20: The translation of internal language types to CPS types.

4.6 CPS conversion

We convert internal language values and valid values to CPS values and valid values using a translation very similar to the elaboration relations given in Section 4.4. The translation of expressions to continuation expressions is higher-order, however; we use continuations in the metalanguage to define the translation itself. In this section that metalanguage is mathematics; in the implementation it is Standard ML, and in Section 4.6.2 it is LF. A continuation-based CPS translation obviates the need for "administrative redices" [112], where the object language's notion of continuation is used instead of the metalanguage's.

Let's first state the translation relation for types, values and valid values, since they are more straightforward. We have relations

$$
\begin{array}{ccc}
[\![A]\!]_{\mathsf{c}} & [\![v]\!]_{\mathsf{vc}}^{A@\mathrm{w}} & [\![s]\!]_{\mathsf{sc}}^{\omega.A} \\
\text{types} & \text{values} & \text{valid values}
\end{array}
$$

We can state the type translation relation, which guides the others. It appears in Figure 4.20. The only type for which we do anything interesting is the function type. Since continuations in the CPS language do not return, we encode an internal language function as a continuation taking two arguments: the (translated) original argument and a continuation representing "what to do next" with the result.

The translation of continuation expressions is given via a function `convert`:

$$\mathsf{convert}\ \mathcal{K}\ M\ A\ \mathrm{w}$$

It takes an internal language expression M to convert and its type A and world w. It also takes a meta-level continuation $\mathcal{K}$ that describes what to do with the result of M after conversion (it has type $\text{value} \to \text{continuation expression}$) [5]. It is best to see an example. The case for the internal language pair projection operation is

$$
\begin{array}{l}
\mathsf{convert}\ \mathcal{K}\ (\#1\ M)\ A\ \mathrm{w} = \\
\qquad \mathsf{convert}\ \mathcal{K}'\ M\ (A \wedge B)\ \mathrm{w} \\
\qquad \text{where } \mathcal{K}'(\mathrm{v}) = \mathtt{let}\ x = \mathtt{fst}\ \mathrm{v} \\
\qquad\qquad\qquad\qquad \mathtt{in}\ \mathcal{K}(x)
\end{array}
$$

In CPS form, we must first evaluate M to a value and then apply the `let...fst` construct to it. So, we begin by recursively calling convert on M with a new meta-level continuation $\mathcal{K}'$. Here, the syntax "where" is an in-place mathematical definition; it is not part of the CPS language. $\mathcal{K}'$ is defined to take the value, project out the first component and bind it to a variable x. We then pass this variable (a value) to the initial continuation $\mathcal{K}$. Observe that we have to guess the type B in the recursive use of convert. Again, this description of the translation is informal; the real conversion is defined over typing derivations, where the type B can be gleaned from the derivation that M is well-typed.

The full translation for expressions is given in Figure 4.21, and most of the cases follow this same form. The case for function application is interesting; after evaluating M and N to values, M is a continuation that expects an argument and a return continuation. We therefore pair the value resulting from N with a reification of the meta-level continuation as an object-level $\boldsymbol{\lambda}$. For an internal language function value (Figure 4.22) we project the original argument from the argument tuple and then run the body; when it finishes, we project the continuation argument and call it on the result. The translation of other values is completely pointwise. Finally, the internal language `get` construct decomposes into two uses of `go`. First we save the current world's address, and make it valid with `put` so that we can use it from the remote world. We then `go` to the remote world using the supplied address and evaluate the body. The resulting value must be of mobile type, so we `put` it to make it valid as well. Using the saved address, we `go` back and continue with the valid value. In this way we have separated the control and data mobility aspects of `get` into the `go` and `put` constructs completely; now only `go` changes between worlds and only `put` uses the mobile judgment.

To convert a whole program we need an initial continuation to pass to convert. The natural choice is one that ignores its argument and returns the `halt` continuation expression.

4.6.1 Static correctness

We can now state the static correctness of CPS conversion. For values and valid-values the statement is familiar; for expressions it is more complex, due to the higher-order nature of the convert function.

Theorem 43 (Functionality of type translation)
$[\![A]\!]_{\mathsf{c}}$ is defined and unique for all well-formed A

Theorem 44 (Static correctness of value CPS conversion)

If	$\mathcal{D} :: \Gamma \vdash v : A@\mathrm{w}$	*(in the internal language)*
then	$[\![v]\!]^{A@\mathrm{w}}_{\mathsf{vc}} = v'$	*(for some v')*
and	$[\![\Gamma]\!] \vdash v' : [\![A]\!]_{\mathsf{c}}@\mathrm{w}$	*(in the CPS language)*

Theorem 45 (Static correctness of valid value CPS conversion)

If	$\mathcal{D} :: \Gamma \vdash s \sim \omega.A$	*(in the internal language)*
then	$[\![s]\!]^{\omega.A}_{\mathsf{sc}} = s'$	*(for some s')*
and	$[\![\Gamma]\!] \vdash s' \sim \omega.[\![A]\!]_{\mathsf{c}}$	*(in the CPS language)*

convert $\mathcal{K}$ (`val` v) A w = $\mathcal{K}$(v)

convert $\mathcal{K}$ (#1 M) A w = convert $\mathcal{K}'$ M $(A \wedge B)$ w
where $\mathcal{K}'$(v) = `let` x = `fst` v `in` $\mathcal{K}(x)$

convert $\mathcal{K}$ (#2 M) A w = convert $\mathcal{K}'$ M $(A \wedge B)$ w
where $\mathcal{K}'$(v) = `let` x = `snd` v `in` $\mathcal{K}(x)$

convert $\mathcal{K}$ (`let` x = M `in` N) C w = convert $\mathcal{K}'$ M A w
where $\mathcal{K}'$(v) = $[{}^{v}/_{x}]$(convert $\mathcal{K}$ N C w)

convert $\mathcal{K}$ (`unpack` ω, x = M `in` N) C w =
convert $\mathcal{K}'$ M $(\exists\omega.A)$ w
where $\mathcal{K}'$(v) = `let` ω,x = `unpack` v `in` convert $\mathcal{K}$ N C w

convert $\mathcal{K}$ (`leta` x = M `in` N) C w =
convert $\mathcal{K}'$ M $(A$ `at` w′$)$ w
where $\mathcal{K}'$(v) = `leta` x = v `in` convert $\mathcal{K}$ N C w

convert $\mathcal{K}$ (`lets` x = M `in` N) C w =
convert $\mathcal{K}'$ M $(\mathfrak{F}_{\omega}A)$ w
where $\mathcal{K}'$(v) = `lets` u = v `in` convert $\mathcal{K}$ N C w

convert $\mathcal{K}$ (`put` u = M `in` N) C w =
convert $\mathcal{K}'$ M A w
where $\mathcal{K}'$(v) = `put` u = v `in` convert $\mathcal{K}$ N C w

convert $\mathcal{K}$ (`localhost ()`) (w `addr`) w = `let` x = `localhost ()` `in` $\mathcal{K}$(x)

convert $\mathcal{K}$ $(M\langle \mathrm{w}'\rangle)$ $([{}^{\mathrm{w}'}/_{\omega}]A)$ w = convert $\mathcal{K}'$ M A w
where $\mathcal{K}'$(v) = `let` x = v$\langle$w′$\rangle$ `in` $\mathcal{K}$(x)

convert $\mathcal{K}$ $(M\ N)$ B w = convert $\mathcal{K}'$ M $(A \supset B)$ w
where $\mathcal{K}'$(v) = convert $\mathcal{K}''$ N A w
where $\mathcal{K}''$(v′) = `call` v($\langle$ v′, λx.$\mathcal{K}$(x)$\rangle$)

convert $\mathcal{K}$ (`get`[M] N) A w = convert $\mathcal{K}'$ M (w′ `addr`) w
where $\mathcal{K}'(v_a)$ = `let` x_r = `localhost ()` `in`
`put` u_r = x_r `in`
`go`[w′; v_a] convert $\mathcal{K}''$ N A w′
where $\mathcal{K}''$(v) = `put` u = v `in`
`go`[w; **vval** u_r] $\mathcal{K}$(**vval** u)

Figure 4.21: The translation of internal language expressions to CPS expressions. The function convert takes a meta-level continuation from CPS values to CPS expressions, an internal language expression, its type and world. It returns a CPS expression. Variables that do not appear in the input are assumed to be completely fresh.

$$
\begin{array}{rcl}
[\![\langle v_1, v_2\rangle]\!]^{\mathrm{w}@A\wedge B}_{\mathsf{VC}} & = & \langle [\![v_1]\!]^{\mathrm{w}@A}_{\mathsf{VC}}, [\![v_2]\!]^{\mathrm{w}@B}_{\mathsf{VC}}\rangle \\
 & \vdots & \\
[\![\boldsymbol{\lambda} x.M]\!]^{\mathrm{w}@A\supset B}_{\mathsf{VC}} & = & \boldsymbol{\lambda} y.\mathtt{let}\ x = \#1\ y\ \mathtt{in} \\
 & & \quad \mathsf{convert}\ \mathcal{K}\ M\ B\ \mathrm{w} \\
 & & \quad \text{where } \mathcal{K}(v) = \mathtt{let}\ x_k = \#2\ y\ \mathtt{in}\ x_k\ v
\end{array}
$$

Figure 4.22: The translation of internal language values to CPS values. Only the case for functions is interesting, so I give that along with the straightforward example of pair values.

Theorem 46 (Static correctness of expression CPS conversion)

If	$\mathcal{D} :: \Gamma \vdash M : A@\mathrm{w}$		*(*in the internal language*)*
and	for all v such that	$\Gamma \vdash v : [\![A]\!]_{\mathsf{C}}@\mathrm{w}$, $\Gamma \vdash \mathcal{K}(v)\star\mathrm{w}$	
then	$\mathsf{convert}\ \mathcal{K}\ M\ A\ \mathrm{w} = c$		*(*for some c*)*
and	$[\![\Gamma]\!] \vdash c\star\mathrm{w}$		*(*in the CPS language*)*

The convert function takes a meta-level continuation $\mathcal{K}$, so the statement of type correctness for convert must say what continuations are acceptable inputs. We insist that if the input to $\mathcal{K}$ is a well-typed value v, that $\mathcal{K}(v)$ is a well-typed continuation at the same world. If that is the case, then the result of convert will also be well-formed. Note that we implicitly quantify over all mathematical functions $\mathcal{K}$ obeying this form, including ones that do things like inspect the argument or that are not constructive. This is a little metatheoretically fishy and in any case far more than we need to prove the theorem. When we formalize this theorem statement in Twelf, we will have a precisely defined language in which to specify $\mathcal{K}$ (LF), and thus will be able to circumscribe the forms it may take (for example, that it be parametric in its argument). This will be especially important because we unify the translation and its proof of static correctness; we care *how* it produces the typing derivation and not simply *that* it produces a typing derivation.

The formal proofs of these theorems appear in Appendix A.8.6 and are discussed in the next section.

4.6.2 CPS conversion in Twelf

In order to define CPS conversion in Twelf we will make use of the Twelf definitions for the CPS language in Figure 4.23 and the IL in Figure 4.13 in Section 4.4.1. We have LF type families for CPS types, values, continuation expressions, and valid values. We have judgments for the well-formedness of these syntactic classes (except for types, which are always well-formed). We also have a type translation function `ttoct` ("type to ctype") which converts IL types to CPS types, and `ttoctf`, which converts a world-parameterized type of the form $\omega.A$.

Declaration	Meaning
`ctyp : type`	CPS types
`cval : type`	CPS values
`cexp : type`	Continuation expressions
`cvval : type`	CPS valid values
`cof : cexp -> world -> type`	Well-formedness of continuation expressions
`cofv : cval -> ctyp -> world -> type`	Well-formedness of values
`cofvv : cvval -> (world -> ctyp) -> type`	Well-formedness of valid values
`ttoct : typ -> ctyp -> type`	Translation of IL types to CPS types
`ttoctf : (world -> typ) -> (world -> ctyp) -> type`	Translation of types parameterized by world

Figure 4.23: Twelf type families defining the CPS language.

The translation for values is a three-place relation, as it was for elaboration. It is defined as follows:

```
tocpsv+ : {WV  : ofv V A W}
          {CT  : ttoct A CA}
          {WCV : cofv CV CA W}
          type.
%mode tocpsv+ +WV +CT -WCV.
%mode tocpsv- +WV -CT -WCV.
```

As before, there are actually two mutually recursive relations (`ttocpsv+` and `ttocpsv-`), which differ only in their modes. Similarly, we have a translation for valid values:

```
tocpsvv- : {WV : ofvv V Af}
           {CT : ttoctf Af CAf}
           {WCV : cofvv CV CAf}
           type.
%mode tocpsvv- +WV -CT -WCV.
```

The validity judgment for valid values is with respect to a world-indexed type, so we use the `ttoctf` translation relation. Because the structure of valid values is so limited, we will only need the "-" version of this relation.

The translation relation for expressions is intricate. I will step through it carefully:

```
tocps- : {M  : exp}
         {WM : of M A W}
         {CT : ttoct A CA}
         % output is parameterized by K
         {CC : (cval -> cexp) -> cexp}
```

```
          % well-formedness of output is
          % parameterized by well-behavedness
          % of K
          {WCC :
           {K : cval -> cexp}
           ({cv : cval}
            {wcv : cofv cv CA W}
            cof (K cv) W) ->
           cof (CC K) W}
          type.
%mode tocps- +M +WM -CT -CC -WCC.
%mode tocps+ +M +WM +CT -CC -WCC.
```

The first three arguments are straightforward: the IL expression to convert, the derivation of its well-formedness, and the translation of its IL type to a CPS type. (This third argument is either an output or an input depending on whether this is `tocps-` or `tocps+`.) The two outputs are higher-order, like the convert function and its statement of type correctness from the previous section. The output `CC` is a continuation expression that is parameterized by what we called $\mathcal{K}$; here it is an LF function that produces a continuation expression from the value representing the result of evaluating M. For example, if we CPS-convert the IL expression `localhost`, `CC` will be the LF term

$$\lambda k : (\texttt{cval} \rightarrow \texttt{cexp}).\ \texttt{clocalhost}(\lambda x : \texttt{cval}.\ k\ x)$$

(Recall that in the CPS language, `localhost` takes the form $\texttt{let}\ x = \texttt{localhost}()\ \texttt{in}\ c$. The LF constant `clocalhost` has type $(\texttt{cval} \rightarrow \texttt{cexp}) \rightarrow \texttt{cexp}$, using higher-order abstract syntax in the standard way to encode the binding of x within c.) If we apply `CC` to the standard initial $\mathcal{K}$, $\lambda y : \texttt{cval}.\ \texttt{halt}$, we get

$$\texttt{clocalhost}(\lambda x : \texttt{cval}.\ \texttt{halt})$$

as expected.

`CC`, the output of the translation, must be also be well-formed. Since it is parameterized by $\mathcal{K}$, the well-formedness of `CC` is contingent upon the well-behavedness of $\mathcal{K}$. Therefore the output `WCC`, representing the well-formedness of `CC`, has several nested implications. First, we quantify over all `K`. Then, assuming some value `cv` and derivation that is well-formed, `wcv`, `K` applied to that value must be well-formed. If that is true, then `CC` applied to `K` will also be well-formed.

The type of this relation may be more clear when we see how it is used. But before we are able to give the translation for expressions and values, we need to prove some lemmas about the type translation relation.

Lemmas

First, we define a shallow (non-inductive) equality relation on continuation types, which internalizes LF equality:

```
ceqtyp : ctyp -> ctyp -> type.
ceqtyp_ : ceqtyp A A.
```

We can then prove functionality and effectiveness for the type translation relation `ttoct`. Functionality means that the output is deterministic, and effectiveness that the translation can be performed for any input.

```
ttoct_fun : ttoct A A' -> ttoct A A'' -> ceqtyp A' A'' -> type.
%mode ttoct_fun +X +Y -Z.
ttoct_gimme : {A:typ} {A':ctyp} ttoct A A' -> type.
%mode ttoct_gimme +A -A' -D.
```

(We have similar lemmas for the `ttoctf` relation, not shown here.) We need to prove that continuation value typing respects equality, and that partial continuation typing (the `WCC` output of `tocps-` above) respects equality. These theorems are trivial because equality is shallow.

```
cofv_resp : cofv C A W -> ceqtyp A A' -> cofv C A' W -> type.
%mode cofv_resp +COF +EQ -COF'.

wcc_resp : {WCC :
            ({K : cval -> cexp}
             ({cv : cval}
              {wcv : cofv cv A W}
              cof (K cv) W) ->
             cof (CC K) W)}

           {EQ : ceqtyp A A'}

           {WCC' :
            ({K  : cval -> cexp}
             ({cv : cval}
              {wcv : cofv cv A' W}
              cof (C cv) W) ->
             cof (CC K) W)}

           type.
%mode wcc_resp +K +EQ -K'.
```

The proofs of these lemmas are uninteresting and appear in Appendices A.8.4 and A.8.6.

Translation

We can now give the translation for expressions. We start by proving the `tocps+` version, since it is only one case:

```
tocps+/- : tocps+ V WV (CTi : ttoct A A') CC WCC
        <- tocps- V WV (CTo : ttoct A A'') CC WCC'
```

```
        <- ttoct_fun CTo CTi (EQ : ceqtyp A' A'')
        <- wcc_resp WCC' EQ WCC.
```

This looks almost the same as the analogous case from elaboration. We receive a derivation of the type translation `CTi` as input, and appeal to the `tocps-` version immediately, which gives us another type translation `CTo`. We then use functionality of `ttoct` to see that `CTo` and `CTi` give the same result, and use the fact that `WCC` respects equality of types to get a `WCC'` that mentions `A'` instead of `A''`, as required.

The translation of the IL projection `fst` is illustrative.

```
c_fst : tocps- (fst M) (&E1 WM) CT
          % parameterized expression resulting from translation
          ([k:cval -> cexp]
             CC ([v:cval] cfst v ([a:cval] k a)))
          % its parameterized typing derivation
          ([k : cval -> cexp][wk : ({cv : cval}
                                    {wcv : cofv cv CA W}
                                      cof (k cv) W)]
             WCC _ ([v][wv] co_fst wv wk))
    <- tocps- M WM (ttoct/& CT _) CC WCC.
```

To translate the expression `fst M`, we inductively translate the argument. It must have type `A & B` and so the only case for translating it is `ttoct/&`, so this subgoal covers all outputs. It returns the translation for `A`, called `CT`, and the translation of `M` and its well-formedness, called `CC` and `WCC` respectively. Our job is now to build the continuation expression for the `fst` projection and its typing derivation. The expression is parameterized by `k`, which takes the result of the `fst` operation. Its body works by first calling `CC` (the translation of `M`) and supplying it with a continuation that binds the result of `M` to the variable `v`. We then project the first component using `cfst`, and apply the outer continuation `k` to the result.

The parameterized typing derivation follows the same plan. It takes a continuation `k` and a typing derivation for it, `wk`. The derivation starts with the typing for the translation of `M`, which is represented by the function `WCC`. We apply `WCC` to the actual continuation we supply above—Twelf can deduce what this is by unification, so we just write `_` to avoid repeating ourselves. The second argument is the typing derivation for that continuation; it takes a variable `v` representing the result of evaluating `M` and a typing derivation for it `wv`. The derivation consists simply of the typing rule for `c_fst` applied to the well-formedness of its argument and the code that follows, both of which we get from arguments.

The translation is challenging because it is so high-order (typing responsibilities pass from callee to caller and vice versa) but most of the cases follow the same pattern. Twelf's term reconstruction allows us to supply only the essence of the translation and it can often determine the rest. In particular, because typing derivations are indexed by the terms that they type, we can usually perform the translation on typing derivations and this will induce the appropriate translation for terms automatically. For example, we can write the case for application as follows:

```
c_app : tocps- (app M N) (=>E WM WN) CTB _
        ([c][wc]
         % eval function, then argument
         FM _ ([f][wf]
               FN _ ([a][wa]
                     co_call wf
                      (cov_pair wa (cov_lam ([r][wr] wc r wr)) ))))
   <- ttoct_gimme (A => B) (A' c& (B' ccont) ccont) (ttoct/=> CTA CTB)
   <- tocps+ M WM (ttoct/=> CTA CTB) _ FM
   <- tocps+ N WN CTA _ FN.
```

The `CC` output is just `_`, and is recovered by Twelf from the `WCC` output (`[c][wc] ...`). The first subgoal in this case (`ttoct_gimme`) exists to reconcile the various type translations that will occur: We will have a translation for `A`, `B`, and `A => B` which all must agree. We therefore invoke the effectiveness lemma on the largest type (`A => B`) and the others will be subterms. We then invoke `tocps` inductively to translate the function and argument expressions (we use the "+" version here so that we need not do equality reasoning about the type translations). The resulting typing derivation begins with the derivations for the translation of `M` and `N`. Given these, it builds a pair of the argument `wa` and return continuation. The body of the return continuation is the outermost continuation passed to the translation of the `app`. We then end with a call to the translated function value on the pair we created.

Binding

Constructs with binders require us to embed the case for variables within the subgoal. For example, the translation of the IL `let` construct is as follows:

```
c_let : tocps- (let M N) (oflet WM WN) CTN _
          ([c][wc] FM _ ([v][wv] FN v wv c wc))
     <- ttoct_gimme A A' CTM
     <- tocps+ M WM CTM _ FM
     <- ( {x}{xof : ofv x A}
          {x'}{x'of : cofv x' A'}
          {thm:tocpsv- xof CTM x'of}
         tocps- (N x) (WN x xof) CTN (CC x') (FN x' x'of)).
```

This translation begins as before, by translating `M`. We then want to translate the body, `N`, but it has type `val -> exp`, so it must be in a subgoal with a hypothetical value variable in context. The subgoal actually introduces five hypotheses: the direct style value `x`; a derivation that it is well-formed at type `A`, `xof`; the CPS value it will be translated to, `x'`; its derivation `x'of`; and the case of the theorem that relates the two. Once we have translated `N` and `WN` in this context, we build the result typing derivation. Because we have set up the translation such that the continuation always takes a value as an argument, we do not need a CPS-level let construct; we simply invoke `FM`, which types the translation of `M`, and then invoke `FN` on the value and typing derivation resulting from that, and finish with the outermost continuation.

The other cases follow these same patterns or techniques already used in elaboration. They can be found in full in Appendix A.8.6. In this formalism, the type-correctness of the translations is manifest in the fact that they translate typing derivations. The termination of the translations and their definedness for all well-typed inputs comes from the Twelf totality assertions. □

4.7 Closure conversion

After converting to the CPS language, the code is at a fairly low level. Each continuation ($\boldsymbol{\lambda}$) body is a sequence of primitive operations on values, ending with a call to some other continuation (or a `halt`). However, in order to implement the semantics not by substitution but with environments (which is much more efficient), we need to make the creation of closures explicit. After doing this, every continuation body will be closed code, and so we can hoist them out to the topmost level and give them global names. I do not formalize this hoisting process, but its implementation is described in Section 5.4.8. Therefore the closure conversion translation will be the last formalized step of compilation before I shift focus to the actual implementation in the next chapter.

To perform closure conversion, we will define a variant of the CPS language that changes the typing rule for $\boldsymbol{\lambda}x.c$ to insist that c have no modal or valid variables in scope other than x. We will also modify the `go` construct, which transfers control to another world and asks it to execute a continuation. In order to represent this continuation (which may have free variables), it will be treated as a zero-argument function and closure converted. The `go` construct will the be replaced with `go_cc`, which takes an address and closure to execute at the remote world.

Closure conversion here is conceptually standard: For each lambda, we will generate a record (the environment) that collects all of the free variables of the function. This environment will be passed as an additional argument to the function, which reconstitutes its free variables by projecting them from the environment. In the modal setting this is complicated by the presence of valid hypotheses and hypotheses at worlds other than the current world. For instance, the following (IL) function at world $\mathbf{w}_0$ has a free variable at $\mathbf{w}_1$:

$$a{:}\mathbf{w}_1\ \texttt{addr@}\mathbf{w}_0, x{:}\texttt{int @}\mathbf{w}_1 \vdash \boldsymbol{\lambda}y.(y + \texttt{get}[\mathbf{w}_1; a]\ x) : \texttt{int} \supset \texttt{int@}\mathbf{w}_0$$

To generate an environment for this function, we need to reify the remote hypothesis as a value. Because it does not make sense here, we do this by using the `at` modality. Similarly, valid hypotheses are encapsulated using the $\mathfrak{B}_\omega$ modality.

A continuation of type A `cont` could have any set of free variables, and therefore can have an arbitrary environment type. It is important, however, that the translation of the CPS type A `cont` to a CC type is only a function of A. We achieve this by hiding the environment type using an existential type [78, 83]. The translation of the type A `cont` is thus

$$[\![A\ \texttt{cont}]\!]_{\texttt{CC}} = \exists\alpha_{\text{env}}.\alpha_{\text{env}} \times (([\![A]\!]_{\texttt{CC}} \times \alpha_{\text{env}})\ \texttt{cont})$$

$$
\begin{array}{rrcl}
\text{types} & A & ::= & \ldots \mid \alpha \\
\text{values} & v & ::= & \ldots \\
 & & \mid & \mathbf{pack}\ B, v\ \mathbf{as}\ \exists\alpha.A \\
\text{valid vals} & s & ::= & \ldots \\
\text{conts} & c & ::= & \ldots \\
 & & \mid & \cancel{\texttt{go}[\mathrm{w}; v_a]\ c} \\
 & & \mid & \texttt{go\_cc}[\mathrm{w}; v_a]\ v_c \\
 & & \mid & \texttt{let}\ \alpha, x = \texttt{unpack}\ v\ \texttt{in}\ c
\end{array}
$$

$$
\frac{\Gamma \vdash v_a : \mathrm{w}'\ \texttt{addr@w} \quad \Gamma \vdash v_k : \exists\alpha_{\text{env}}.(\alpha_{\text{env}} \wedge (\texttt{unit} \wedge \alpha_{\text{env}})\ \texttt{cont})\texttt{@w}'}{\Gamma \vdash \texttt{go\_cc}[\mathrm{w}'; v_a]\ v_k\texttt{@w}}\ \text{go\_cc}
$$

$$
\frac{\cdot, x : A\texttt{@w} \vdash c\star\mathrm{w}}{\Gamma \vdash \boldsymbol{\lambda} x.c : A\ \texttt{cont@w}}\ \boldsymbol{\lambda}
$$

$$
\frac{\Gamma \vdash v : [{}^{B}/_{\alpha}]A\texttt{@w}}{\Gamma \vdash \mathbf{pack}\ B, v\ \mathbf{as}\ \exists\alpha.A : \exists\alpha.A\texttt{@w}}\ \mathbf{pack}\ \alpha
$$

$$
\frac{\Gamma \vdash v : \exists\alpha.A\texttt{@w} \quad \Gamma, \alpha\ \mathsf{type}, x{:}A\texttt{@w} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{let}\ \alpha, x = \texttt{unpack}\ v\ \texttt{in}\ c\star\mathrm{w}}\ \texttt{unpack}
$$

Figure 4.24: The closure converted (CC) language, defined as a modification of the CPS language in Figures 4.17, 4.16, 4.15, and 4.14.

No other types are affected by closure conversion, so the translation $[\![\cdot]\!]_{\mathsf{cc}}$ is pointwise there.

The CC language is given in Figure 4.24 as a modification of the CPS language. We add type variables and existential types to support the translation of the continuation type. We remove `go` and replace it with `go_cc`; its argument is a closure-converted `unit cont`. Finally, the typing rule for $\boldsymbol{\lambda}$ requires it to be closed. This makes the property of the translation producing proper closures a matter of typing.

The translation of the program syntax is as follows. We only translate $\boldsymbol{\lambda}$ values, `call` and `go` expressions; every other construct is translated in a pointwise fashion. The `call` construct is a good starting point because it shows how closures are used:

$$\begin{array}{rcl}
[\![\texttt{call}\ v_f(v_x)]\!]_{\mathsf{cc}} & = & \texttt{unpack}\ [\![v_f]\!]_{\mathsf{cc}}\ \texttt{as}\ \langle \alpha_{\mathrm{env}}, p\rangle\ \texttt{in} \\
& & \texttt{let}\ e = \texttt{fst}\ p\ \texttt{in} \\
& & \texttt{let}\ f = \texttt{snd}\ p\ \texttt{in} \\
& & \texttt{call}\ f\langle [\![v_x]\!]_{\mathsf{cc}}, e\rangle
\end{array}$$

We start by unpacking the closure, to get the environment type and the pair value. Its first component is the environment and the second is the function; we project these out. We then call the function with a pair of arguments: the translated argument and the extracted environment.

The `go` construct is translated by handing the work off to the case for $\boldsymbol{\lambda}$. It is as follows:

$$[\![\,\mathsf{go}[\mathrm{w}; v_a]c\,]\!]_{\mathsf{cc}} \quad = \quad \mathsf{go\_cc}[\mathrm{w}, [\![v_a]\!]_{\mathsf{cc}}]\ [\![\,\boldsymbol{\lambda} y.c\,]\!]_{\mathsf{cc}}$$

We assume the variable y to be fresh; it has type `unit` and is unused.

The translation of continuations is the crux of closure conversion:[1]

$$
\begin{array}{rcl}
[\![\boldsymbol{\lambda} x.c]\!]_{\mathsf{CC}} & = & \mathbf{pack}\ \langle B_{\text{env}}, \langle v_{\text{env}}, \boldsymbol{\lambda} p.c' \rangle \rangle \\
 & & \mathbf{as}\ \exists \alpha_{\text{env}}.\alpha_{\text{env}} \wedge ((([\![A]\!]_{\mathsf{CC}} \wedge \alpha_{\text{env}})\ \mathtt{cont}) \\
\text{where}\ldots & & \\
c' & = & \mathtt{let}\ x = \#2\ p\ \mathtt{in} \\
 & & \mathtt{let}\ e = \#1\ p\ \mathtt{in} \\
 & & \mathtt{leta}\ \mathsf{FV}_1 = \pi_1 e\ \mathtt{in} \\
 & & \qquad\vdots \\
 & & \mathtt{leta}\ \mathsf{FV}_n = \pi_n e\ \mathtt{in} \\
 & & \mathtt{lets}\ \mathsf{FSV}_1 = \pi_{n+1} e\ \mathtt{in} \\
 & & \qquad\vdots \\
 & & \mathtt{lets}\ \mathsf{FSV}_m = \pi_{n+m} e\ \mathtt{in} \\
 & & [\![c]\!]_{\mathsf{CC}} \\
v_{\text{env}} & = & \langle \mathbf{held}\ \mathsf{FV}_1,\ \ldots,\ \mathbf{held}\ \mathsf{FV}_n, \\
 & & \ \mathbf{sham}\ \omega.\mathsf{FSV}_1,\ \ldots,\ \mathbf{sham}\ \omega.\mathsf{FSV}_m \rangle \\
B_{\text{env}} & = & [\![\mathsf{FVT}_1]\!]_{\mathsf{CC}}\ \mathtt{at}\ \mathsf{FVW}_1 \wedge \ldots \wedge [\![\mathsf{FVT}_n]\!]_{\mathsf{CC}}\ \mathtt{at}\ \mathsf{FVW}_n \wedge \\
 & & \ \mathfrak{B}_\omega \mathsf{FSVT}_1 \wedge \ldots \wedge \mathfrak{B}_\omega \mathsf{FSVT}_m \\
n & = & \text{Number of free modal variables in } c. \\
\mathsf{FV}_i & = & \text{The } i^{\text{th}} \text{ free modal variable of } c. \\
\mathsf{FVT}_i & = & \text{The type of the } i^{\text{th}} \text{ free modal variable of } c. \\
\mathsf{FVW}_i & = & \text{The world of the } i^{\text{th}} \text{ free modal variable of } c. \\
m & = & \text{Number of free valid variables in } c. \\
\mathsf{FSV}_i & = & \text{The } i^{\text{th}} \text{ free valid variable of } c. \\
\mathsf{FSVT}_i & = & \text{The type of the } i^{\text{th}} \text{ free valid variable of } c, \\
 & & \text{parameterized by } \omega.
\end{array}
$$

The continuation body c' begins by projecting from its pair argument the real argument x and the environment e. It then extracts each of the free variables in sequence; first the free modal variables ($x{:}A@\mathrm{w}'$) and then the free valid variables ($u \sim \omega.A$). The modal variables are encapsulated by the `at` modality so we use `leta` to bind them; the valid variables by $\mathfrak{B}$ and so we use `lets` to bind those. The type of the environment reflects this representation; it is an iterated conjunction of $A_i\ \mathtt{at}\ \mathrm{w}_i$ and then $\mathfrak{B}\omega A_j$.

An interesting observation is how the process of closure conversion exercises the expressiveness of the programming language. In order to do it, we must be able to encapsulate any kind of dynamic hypothesis into a value in order to store it in the environment, and then restore that hypothesis into the context within the body of the closure. This is a kind of completeness criterion akin to the identity property for sequent calculi [106]. At the source level it means that any piece of code can be hoisted out of its context by abstracting over its free variables, an activity that is common when

[1] To simplify the presentation we assume derived forms for iterated products, where $\langle v_1, v_2, \ldots, v_n \rangle$ is $\langle v_1, \langle v_2, \langle \ldots, v_n \rangle \rangle \rangle$, and π_i is a projection of the i^{th} component.

programming. In fact, during the development of this work, we originally omitted the `at` connective and used a less precise version of the ⟜ connective. Our inability to do closure conversion was what alerted us to their expressiveness (and necessity!).

The static correctness of closure conversion is stated in the familiar way:

Theorem 47 (Functionality of CC type translation)
$[\![A]\!]_{\mathsf{cc}}$ is defined and unique for all well-formed A

Theorem 48 (Static correctness of value closure conversion)

If	$\mathcal{D} :: \Gamma \vdash v : A@\mathrm{w}$	*(in the CPS language)*
then	$[\![v]\!]_{\mathsf{ccv}} = v'$	*(for some v')*
and	$[\![\Gamma]\!] \vdash v' : [\![A]\!]_{\mathsf{cc}}@\mathrm{w}$	*(in the CC language)*

Theorem 49 (Static correctness of valid value closure conversion)

If	$\mathcal{D} :: \Gamma \vdash s \sim \omega.A$	*(in the CPS language)*
then	$[\![s]\!]_{\mathsf{ccs}} = s'$	*(for some s')*
and	$[\![\Gamma]\!] \vdash s' \sim \omega.[\![A]\!]_{\mathsf{cc}}$	*(in the CC language)*

Theorem 50 (Static correctness of expression closure conversion)

If	$\mathcal{D} :: \Gamma \vdash c\star\mathrm{w}$	*(in the CPS language)*
then	$[\![c]\!]_{\mathsf{cc}} = c'$	*(for some c')*
and	$[\![\Gamma]\!] \vdash c'\star\mathrm{w}$	*(in the CC language)*

The typing condition on $\boldsymbol{\lambda}$ in the CC language ensures that the output is indeed closure-converted. Each of these theorems is an easy induction. For `go`, we appeal immediately to induction on $\boldsymbol{\lambda}y.c$; this is well-founded because we consider the induction metric to be lexicographic in the number of occurrences of `go` and then the size of the term. For `call`, after applying induction it is a simple matter of observing that the `unpack`, projections, and `call` are well-typed.

In the case for $\boldsymbol{\lambda}x.c$, we have $[\![\Gamma]\!], x{:}[\![A]\!]_{\mathsf{cc}}@\mathrm{w} \vdash [\![c]\!]_{\mathsf{cc}}\star\mathrm{w}$ by induction hypothesis. We then strengthen this to $\mathsf{FV}, \mathsf{FSV}, x{:}[\![A]\!]_{\mathsf{cc}}@\mathrm{w} \vdash [\![c]\!]_{\mathsf{cc}}\star\mathrm{w}$ where FV and FSV are the sets of actually occurring modal and valid variables. In the translation, we discharge each of these hypotheses by the series of `lets` and `leta` bindings wrapping the translated c, leaving us with only the hypothesis $p{:}[\![A]\!]_{\mathsf{cc}} \wedge B_{\text{env}}$ representing the argument. This meets the typing condition for closed lambdas. It is easy to see that $v_{\text{env}} : B_{\text{env}}$ and therefore that the `pack` is well-formed. □

As usual, the formal statement and proof of the type correctness of closure conversion is carried out in Twelf. The reader should note that this proof is the least similar to the hypothetical paper version, because we encode an essentially different closure conversion algorithm and introduce a special syntactic form for the representation of closures. This is to keep the complexity of the proof manageable while still getting at its essence; it is nonetheless the largest Twelf proof in this project. I describe the technique in the next section.

4.7.1 Closure conversion in Twelf

Higher order abstract syntax

Most Twelf encodings use the technique of *higher order abstract syntax* (HOAS) [108] to represent binders in an object language using the binding structure of LF. This technique has many advantages, such as often getting object language mechanisms and metatheory related to binding (such as substitution and its well-behavedness) "for free."

Sometimes the object language's notion of binding or substitution does not coincide with LF's. For example, in Section 3.4 we had a higher-order substitution operation for falsehood variables that had to be described explicitly and for which we had to prove a substitution theorem. This is seldom any trouble because substitution theorems are generally easy to prove. It can also be the case that we place additional restrictions on the occurrence of variables. For example, to express linear logic [46] we can use LF's binding structure to encode variables but also impose an additional restriction (enforced via a judgment) that the appearance of these variables follow the rules of linear logic [24]. We define a relation

```
linear : (val -> exp) -> type.
```

that takes an expression $x.e$ with one free variable and insists that the variable is used linearly within that expression. In the well-formedness judgment for terms, we require for each linear variable bound that it occurs linearly within the body. For example, the case for functions might be

```
oflam : ofv (lam ([x:val] M x)) (A -o B)
     <- ({x:val} ofv x A ->
           of (M x) B)
     <- linear M.
```

We then need to prove that operations like substitution preserve linearity, but these are straightforward properties.

We can sometimes encounter trouble with HOAS encodings because of the structured way in which we interact with the context in Twelf. For these, we can always resort to a first-order representation where contexts are represented explicitly, but then it becomes very tedious to manipulate terms and prove metatheory—at this level of detail, many things that we take for granted (such as the commutation of bindings, weakening, equality reasoning, etc.) become a substantial fraction of the work. (Fortunately, it is possible to convert to explicit representations of various degrees only for a local portion of a proof [66, 67].) We wish to avoid this as much as possible.

The heart of closure conversion is computing the free variable set of a function. This operation is easy to define on paper but difficult when using a HOAS representation in Twelf because we have no easy way of identifying those terms that are actually variables. In order to implement closure conversion without resorting to explicit contexts, we instead redefine the algorithm so that it does not need to compute free variable sets. We do this by orienting closure conversion not around lambdas themselves but around the sites that bind variables. The resulting algorithm is not one that we would want to use in a compiler (its performance is quadratic in the size of the term), but it produces

the same output. This technique is due to Karl Crary.

Closure converted language

We begin by defining the CC language in Twelf. We have

```
ccexp  : type.
ccval  : type.
ccvval : type.
```

and reuse `ctyp` from CPS for CC types. Most of the syntax of the language is the same as in CPS, however, we modify the `go` construct and replace $\boldsymbol{\lambda}$ with a closure value:

```
ccgo : world -> ccval -> ccval -> ccexp.
ccclosure : (ccval -> ccval -> ccexp) -> ccval -> ccval.
```

The `ccclosure` construct encapsulates the idiomatic way in which we construct closures in the previous section; the closure

$$\begin{array}{l}\textbf{pack}\ B, \langle v_{\text{env}}, \boldsymbol{\lambda} p.\texttt{let}\ x = \#1\ p\ \texttt{in}\ \texttt{let}\ e = \#2\ p\ \texttt{in}\ c\rangle \\ \textbf{as}\ \exists\alpha.\alpha \wedge ((A \wedge \alpha)\ \texttt{cont})\end{array}$$

is instead represented as

$$\textbf{closure}\ x, e.c\ \textbf{with}\ v_{env}$$

where x and e are bound within c. We regard the closure as having type A `cont`. This means that the type translation from the CPS language to the CC language is the identity, which saves us some work in this proof.

The typing conditions for the language are interesting, because they ensure that closures are indeed closed. Rather than put a condition on the typing rule for `ccclosure` (it is "too late" to do so) we instead put a condition on each variable binding site in the language. This is similar to the `linear` judgment in the example above:

```
frozen  : (ccval -> ccexp) -> type.
vfrozen : (ccval -> ccval) -> type.
```

An expression with a free variable $x.c$ is frozen if x does not appear within the body of a `ccclosure`. It may appear anywhere else, including the environment part of a closure. It is easy to specify this in Twelf; the case for closures is:

```
vf/closure : vfrozen ([x] ccclosure ([a][e] BOD a e) (ENV x))
          <- vfrozen ENV.
```

The variable `x` is the variable in question. Because the existential variable `ENV` is applied to it, it may appear there (but must be recursively frozen). However, `BOD` is not applied to `x` and therefore `x` cannot occur in it. A variable is also frozen within any expression or value if it does not occur at all; this will later allow us to create environments that contain *only* the free variables and no others:

```
vf/closed : vfrozen ([x] V).
```

We use the `frozen` family of judgments as a well-formedness condition for every binder. For example, the typing rule for the first projection operation is

```
cco_fst : ccofv V (A c& B) W ->
          ({v}{ov : ccofv v A W} ccof (C v) W) ->
          frozen C ->
          ccof (ccfst V C) W.
```

We also have relations for the frozenness of modal variables within valid values, and for the frozenness of valid variables within values, expressions, and valid values.

During translation of the CPS `cfst` to the CC `ccfst`, we will first recursively translate the body, which may contain `clam`s that become `ccclosure`s. Since `cfst` binds a variable, we then crawl over the term to make sure it doesn't appear in any closures; we do this by modifying any closure where it does to store the variable in its environment and project it within the body. We extract from this traversal a derivation of the variable's frozenness, which we use to construct the well-typedness of the `ccfst`.

We therefore define a function that freezes a variable within an expression (and value, and valid value):

```
freeze : {N  : ccval -> ccexp}
         {N' : ccval -> ccexp}
         {F  : frozen N'}
         type.
%mode freeze +N -N' -F.
vfreeze : {N  : ccval -> ccval}
          {N' : ccval -> ccval}
          {F  : vfrozen N'}
          type.
%mode vfreeze +N -N' -F.
```

It takes an expression with a free variable, and returns a new expression (with a free variable) and a derivation of its frozenness. There are two interesting cases. The first is when the variable does not occur; we then do nothing:

```
fz/closed : vfreeze ([v] V) ([v] V) vf/closed.
```

This is important so that we do not put all bound variables in every environment. We place this case first in the logic program so that we always prefer it over the others.[2] The other case is where we reach a closure:

```
fz/closure :
  vfreeze ([x:ccval] ccclosure
                       ([a:ccval][e:ccval] BOD a e x) (ENV x))
          ([x] ccclosure
               ([a:ccval][e:ccval]
                ccfst e [exh:ccval]
                ccsnd e [envtail:ccval]
                ccleta exh [ex:ccval]
                BOD' a envtail ex)
```

[2]Interestingly, however, the metatheorem holds for any order of the clauses, so we know that a whole range of strategies are sound, from the most conservative (include every bound variable) to the one that logic search returns first (include only the actually occurring variables).

```
              (ccpair (ccheld x) (ENV' x)))
    (vf/closure (vf/pair (vf/held vf/var) FENV))
<- ({a:ccval}{e:ccval}
    freeze ([x] BOD a e x) ([x] BOD' a e x) _)
<- vfreeze ENV ENV' FENV.
```

To freeze a variable `x` within a closure, we first freeze it recursively within the environment part, where it is allowed to appear as usual. We then recursively freeze it within the body—it may appear there as well (inside nested closures). Given the translated body and environment, we then construct a new closure whose body is closed with respect to `x`. We construct a new environment, which is the pair of `x` and the old (translated) environment. The variable `x` is `held` because it may be typed at another world (to freeze a valid variable, we use the shamrock modality). The body of the closure takes an argument (which we leave untouched) and the environment, which will now have `held x` as its first component. Inside the body we project out this value, bind it with `leta`, and pass it with the argument and the tail of the environment to the translated body. In this way, we incrementally build up the closure's environment for each variable binding we encounter during translation.

To CPS convert the program, we will convert each lambda to a closure with an empty environment, and perform the freezing procedure for each binder we see. This translation will be given on typing derivations as usual, so we will first need a number of easy lemmas about the well-behavedness of freezing.

Lemmas

The first family of lemmas state that freezing preserves the closedness of a term:

```
permaclosed :
      {F : {v:ccval} freeze ([x] N x) ([x] N' v x) (Z v)}
      {E : {v} {x} ccexp-eq (N'' x) (N' v x)}
      type.
%mode permaclosed +F -E.
```

This lemma states that if, in some context where there is a bound variable `v`, we freeze $x.N$ (which does not mention `v`) to get $x.N'$, that N' does not mention `v` either. We state this by returning an equality between N' and an N'' where N'' can not mention `v`. Unfortunately we need many versions of this lemma: $n * f$ where n is the number of kinds of variables that might be bound (two: valid or mobile), and f is the number of freezing operations we have (six: n times the number of syntactic classes, which is three). This combinatoric explosion is what accounts for most of the bulk of this proof, even though the lemmas are quite easy. (Polymorphism or automatic theorem proving would help reduce the repetition, if they were to be implemented in Twelf.)

This lemma is used to prove another, that freezing preserves the frozenness of a term:

```
permafrost :
      {ZN : {v:ccval} frozen ([y] N v y)}
      {FN : {y:ccval} freeze ([v] N v y) ([v] N' v y) (F y)}
```

```
      {ZN': {v:ccval} frozen ([y] N' v y)}
      type.
%mode permafrost +ZN +FN -ZN'.
```

Again we need 12 lemmas of this form. We prove that if we have a term frozen with respect to the variable `y`, and we freeze some other variable `y` within it, then the term is still frozen with respect to `y`.

Next, we prove that freezing preserves well-formedness of expressions (and values, and valid values):

```
freeze/ok : {WN  : {x}{xok:ccofv x A W} ccof (N x) W'}
            {D   : freeze N N' F}
            {WN' : {x}{xok:ccofv x A W} ccof (N' x) W'}
            type.
%mode freeze/ok +WN +D -WN'.
```

This lemma is simple: If we have a well formed expression with a free variable, and freeze it, then the result is also well-formed. Finally, we have an effectiveness lemma for each of the freezing procedures:

```
freeze-gimme : {N : ccval -> ccexp} {Z : freeze N N' F} type.
```

Translation

We can now define the translation on typing derivations. Because the translation on types is the identity, it simply takes a CPS typing derivation and returns a CC typing derivation at the same type.

```
cc : {D  : cof  M  W}
     {D' : ccof M' W}
     type.
%mode cc +D -D'.

ccv : {D  : cofv  V  A W}
      {D' : ccofv V' A W}
      type.
%mode ccv +D -D'.
```

To translate a binder like `cfst`,

```
- : cc (co_fst WV WN) (cco_fst WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofv x' A W}{thm : ccv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freeze-gimme M' (Z : freeze _ _ F)
 <- freeze/ok WN' Z WN''.
```

we translate the values and body recursively, getting derivations that they are well-typed, `WV'` and `WN'`. We then use the effectiveness of `freeze` to see that we can freeze the bound variable within the body. Using `freeze/ok` we know that this frozen body is also well-typed, so we reassemble the `fst` and we are done.

When we encounter a lambda, we translate it to a closure with an empty environment:

```
- : ccv (cov_lam [x][wx] WM x wx) (ccov_closure ccov_unit
                                     ([x][xof][e][eof] WM'' x xof)
                                     ([e] f/closed)
                                     ([x] FwrtBOD))
 <- ({x}{wx}{x'}{wx' : ccofv x' A W}{thm : ccv wx wx'}
       cc (WM x wx) (WM' x' wx' : ccof (M' x') W))
 <- freeze-gimme M' (Z : freeze _ _ FwrtBOD)
 <- freeze/ok WM' Z WM''.
```

For a closure to be well-formed (constant `ccov_closure`) its environment must be well-formed, the body must be well-formed assuming a well-typed argument and environment, and the body must be frozen with respect to its environment and argument. These are easy to establish because at this point the environment is empty (`unit`) and unused in the body.

Finally, we translate the `go` construct to its closure-converted counterpart. In the last section we took the shortcut of recursing on $\boldsymbol{\lambda} y.c$ where y is an unused variable of type unit. If we did this here we would need a more complex induction metric because `clam ([y] C)` is not a subterm of `cgo W V C`. Instead, we build in the case for lambdas above, specialized to this use:

```
- : cc (co_go WA WC) (cco_go WA' (ccov_closure ccov_unit
                                    ([x][xof][e][eof] WC')
                                    ([x] f/closed)
                                    ([x] f/closed)))
 <- ccv WA WA'
 <- cc WC (WC' : ccof M' W).
```

Here, building the closure is easier because neither the argument nor environment are used within the body.

The rest of the cases follow the same basic pattern, and we can then verify that the relations are total, giving us the desired result. The full proof (which is lengthy mostly because of the lemmas) appears in Appendix A.8.7. □

4.8 Conclusion

In this chapter I have presented an idealization of the first few phases of typed compilation for a modally-typed programming language. We began with the external language and eliminated derived forms by elaborating to a simpler internal language. We then converted to a continuation passing style, which sequences primitive operations and represents the stack behavior of function calls explicitly. Finally, we performed closure conversion to allow us to represent lexically-scoped, nested functions.

We formalized each of these languages in Twelf, as well as the translations between them, and proved the static correctness of those translations. For the internal language and CPS languages we gave dynamic semantics and proved that the type system is

sound. It would be possible to continue further, but as we get deeper into compilation, the languages and algorithms become more difficult to express (such as we found for closure conversion) and less like the implementation we will ultimately use, leading to diminishing intellectual returns on formalization.

Therefore, we now turn our focus to the actual implementation of ML5, which is the subject of the next chapter.

Chapter 5

ML5 and its implementation

ML5 is a programming language for distributed computing [88], based on the modal logics presented in Chapter 3. It is an integration of those primitive constructs into a full-fledged ML-style programming language. Because the language constructs are derived from a propositions-as-types view of logic, they integrate cleanly with ML (which is itself based on a compatible interpretation of propositional logic).

The ML5 implementation is based on the compilation strategy studied in Chapter 4. It type-checks and translates an ML5 source program into low-level code for each of the hosts involved in a computation, and provides a substrate on which that code can be run. Currently, the implementation is specialized to web programming, a particular application of distributed computing where there are exactly two hosts: the web browser and the web server. With only some small changes to the runtime it could be extended to networks of arbitrary size. Therefore, I will discuss the language in its full generality.

In the implementation we are concerned with the usability of the language and with practical considerations of running it on real computers and networks. This will bring up some issues that we have not yet encountered: type and validity inference in the source language, network signatures, exceptions, optimizations to produce more efficient code, the runtime system, and marshaling. These will make the implementation substantially more complex than the idealized compiler from the previous chapter. For this reason the argument for its correctness is informal, and the descriptions of the translations in this chapter are less rigorous. On the other hand, I evaluate the implementation by building applications in it; these are described in Chapter 6.

Finally, this compiler for an ML-like language is the seventh that I have worked on or written. Writing similar programs over and over encourages experimentation, to counter problems experienced in previous iterations and to reduce monotony. For this compiler (particularly in its later phases) I have experimented with ML programming techniques for the development of type-directed compilers. Although these are not specific to the modal setting, I nonetheless spend some time explaining them because I believe them to be worthwhile techniques.

This chapter is organized as follows. I begin by describing the ML5 language by example, using web programming as the application domain (Section 5.1). I then broadly

describe the compiler's architecture and the marshaling strategy, since it has a substantial impact on the way the compiler is designed (Section 5.2.1). After this, I present the implementation following the route that code takes through the compiler. This starts with the front-end (Section 5.3), which comprises the parser, elaborator, and internal language. We then convert to the CPS language (Section 5.4.2). This language has an interesting implementation (Section 5.4.1) and interface for writing type-directed transformations (Section 5.4.3). It is also where most of the relevant work of the compiler takes place, from type representation (Section 5.4.5) to closure conversion (Section 5.4.6) and hoisting (Section 5.4.8) to optimizations (Section 5.4.4). Finally, we generate code for the hosts involved in the computation (Section 5.4.9). The code is supported by a runtime system for the client (Section 5.5.3) and a web server and execution engine (Section 5.5.1) for the server. These parts communicate using a marshaled data format (Section 5.5.4). I summarize in Section 5.6 before presenting some of the applications in the following chapter.

5.1 ML5

ML5 closely resembles Standard ML [77] in syntax and semantics. Its type system is, of course, based upon the modal and validity typing judgments introduced in Chapter 3. For simplicity, I do not include a module system. I believe that the modal type theory is compatible and orthogonal. To ensure that the implementation does not abuse the fact that there are no modules, I support abstract types via the network signature mechanism. This gives a straightforward path to support for separate compilation [129, 130] and suggests how modules could be integrated.

Essentially all the rest of Standard ML is supported, from mutually-recursive functions to pattern matching, datatypes, extensible types and exceptions. I have also made minor changes to excise warts or add incidental features, as is the prerogative of the *auteur*.

ML5 is implemented via elaboration, so to give a semantics at any level of formality requires introducing the intermediate language. Rather than jump into the details, I start with an informal tour of the external language by example. This includes showing what happens when a program is compiled and run. After that, I begin describing the implementation, which includes a more formal account of ML5's type system via elaboration.

5.1.1 Hello, version!

Let us begin with an example program before touring the language systematically.

```
unit
  import "std.mlh"

  extern bytecode world server
  extern val version : unit -> string @ server
```

```
  extern val alert : string -> unit @ home
  extern val server : server addr @ home

  fun showversion () =
      let val s = from server
                  get version ()
      in
          alert [Server's version is: [s]]
      end

  do showversion ()

end
```

Every ML5 program is wrapped in the syntax `unit...end` to delineate it as a compilation unit. Only one compilation unit is currently supported. The body of a compilation unit is a series of declarations. We begin by importing the standard header file that includes a number of common declarations. This includes declarations of the `list`, `order`, and `option` types, as well as primitive operations such as integer math, access to arrays, strings, and references, etc.

We then describe what we need to know about the network by using `extern` declarations; this is the *network signature*. First, we declare a world (host) called `server`. When we attempt to run the program, there must be a host that actually exists called `server` or the program will not be able to run. It will be the runtime system's responsibility to ensure this and to resolve the name "server" to a particular machine. The keyword `bytecode` indicates the *worldkind* of the world, which tells us what kind of low-level code it expects. The worldkind `bytecode` is a simple interpreted language that runs within the ML5 web server (Section 5.5.1); the other available worldkind is `javascript` for producing JavaScript [32] to run in a web browser. The set of available worldkinds is only limited by what code generators are actually available in the compiler, and otherwise has no meaning in the language's semantics.

There must be at least one world in order for the program to mean anything; this world, where the program begins execution, is called `home` and is provided in the initial environment. For web programming, this is the web client (web browser) and so it is a `javascript` world.

Once we have declared the existence of worlds we can declare the existence of resources at those worlds. (It is also possible to declare globally available resources and abstract types. These will be discussed in Section 5.1.3.) The `extern val` declaration asserts the existence of a value with the specified type and world, and binds an ML5 variable to it. The value's name is assumed to be the same as the ML5 identifier, unless the long syntax is used:

```
extern val v : string -> unit @ server = version
```

(Here `version` is a label indicating which resource we are making reference to, and `v` is the ML5 variable bound.) If these resources do not actually exist, then the program will not be able to run; it is the responsibility of the runtime system to resolve them.

We declare that there is a function on the server that returns its version, and that there is a function on the client that displays an alert message to the user. We also declare that on the client we have the address of the server. The namespace for worlds, types, and values are distinct, so the convention is to also call this `server`. To be clear, we have imported a local resource whose label is `server` and bound it to an ML5 variable called `server`; it is an address for the world also called `server`.

Now that we have described what we require of the network, we can write code using those resources. We define a function `showversion` that displays the server's version on the client. The `from` *address* `get` *expression* construct is the `get` construct from Lambda 5. Here `server` is the address of the world we wish to contact, and the expression is a call to the `version` function that exists on the server. We bind the result to a variable at home called `s`. The square brackets are ML5's alternative string literal syntax. As an ML5 expression, square brackets delimit a string; within a string, they delimit an ML5 expression which must have type `string`. These brackets properly nest, so here we build up a string containing the value of `s`. We pass this to `alert` which will display it on the client. The `do` declaration simply evaluates an expression and ignores the result; here we use it to invoke the function on the client to start the program.

Compiling and running a program

The ML5 compiler is called ML5/pgh. We use it to generate the JavaScript and server-side code that implements our application. Supposing the code above is in the file `tests/example.ml5`, we compile it with

```
./ml5pgh tests/example.ml5
```

which produces the files `example_home.js` and `example_server.b5` in the `tests/` directory. The first is the JavaScript code particular to the example; it will be combined with the JavaScript runtime system common to every program (Section 5.5.3) to run on the client's web browser. The second file is the server-side code, which is a type-erased version of the lowest level intermediate language of the compiler (Section 5.4.9).

To run this application, we first ensure that the Server 5 web server (Section 5.5.1) is running on our host (`tom7.org` in this example) and configured to find applications in the `tests/` directory. We then visit a URL like

```
http://tom7.org:7777/5/example
```

in a web browser. (`tom7.org` is the internet host running Server 5; 7777 is the network port, and `example` is the name of the application to launch.) When this URL is visited, the server launches a new instance of the example program and gives it a fresh session identifier. The instance contains a parsed version of the server code, a thread queue for the server (which begins empty), for example. It then sends the common JavaScript runtime to the client along with the session identifier and the code in `example_home.js`,

inside of a tiny stub HTML page. The client runtime creates a network connection with the server that it uses to exchange data with it, and begins executing the application code. For this example, the code calls back to the server and causes it to run some code that calls the `version` resource. The server code then calls to the client and causes it to run code that uses the `alert` resource to display the version string.

5.1.2 Type and validity inference

ML5 has two kinds of bindings for values: modal ($x{:}A@\mathrm{w}$) and valid ($u{\sim}\omega.A$). An important feature of ML5 is type and validity inference, which enables the programmer to syntactically omit types while still enjoying the benefits of static type-checking. Like Standard ML, ML5 infers simple types for local variables (such as function arguments) and polymorphic types for declarations (such as `val` and `fun`). In addition, ML5 infers worlds for these; if the world is unconstrained, then it automatically produces a valid declaration.

For example, the definition of the standard `map` function over lists is as follows:

```
fun map f nil = nil
  | map f (h :: t) = f h :: map f t
```

Like in Standard ML, the function has a polymorphic type, so it can be applied to any type of list. Because the function does not access any local resources (in fact, it is closed) ML5 also allows it to be valid. This means that it can be used at any world:

```
(* ... *)
val l = from server
        get  map (fn x => x + 1) (0 :: 1 :: nil)
```

World inference integrates cleanly into a standard Hindley-Milner approach to type inference, using the same mechanisms. Validity inference is then analogous to polymorphic generalization. The implementation of these is discussed during the description of elaboration (Section 5.3.3).

As is standard, a binding's right-hand side must be a value in order for it to be polymorphically generalized or made valid. The programmer can also bind the result of an expression in the validity context by using `put`:

```
let
  put message = "hello, " ^ "world"
in
  from server
  get  display message
end
```

As it did in our lambda calculi, `put` requires its argument to be of mobile type.

In addition to validity inference and `put`, there are a few other ways that valid bindings can be produced. One of these is the network signature, described next, and others will be mentioned as they are encountered.

5.1.3 Interacting with the environment

Let's extend our example to make use of more interesting local resources, which will exercise the network signature mechanism.

Network signatures and the DOM

We saw how we could declare worlds and import simple resources from them. In the domain of web programming we know the set of worlds (the `server` which runs bytecode and `home`, the client, which runs JavaScript). These worlds are declared for us in the standard header, along with their addresses:

```
extern bytecode   world server
extern javascript world home

extern val server ~ server addr
extern val home   ~ home   addr
```

This means that we do not need to declare the worlds and addresses as we did in the above example (although it does not hurt). The addresses that we import in this example are global resources (indicated by the ~ character to mimic the validity judgment $\sim$); the valid variable `server` is an address that can be used at any world. This is useful because it allows us to write code that access the server, and that can be run in any world because any world has the server's address available.

ML5/pgh also provides some useful libraries that allow access to local resources. An important one is the interface to the web browser's Document Object Model [60]—abbreviated DOM—which is the way of programmatically manipulating the current web page. The DOM is the abstract syntax of the web page, represented as a hierarchical tree of (optionally named) nodes. For example, Figure 5.1 shows HTML source code and a (simplified) DOM tree for it. To define an interface to the DOM, we declare the existence of an abstract type of nodes:

```
extern type dom.node = lc_domnode
```

This binds the ML5 identifier `dom.node` to an abstract type, imported from the environment. Here we use the long form where we name the label of the type we are importing, because labels (eventually compiled to JavaScript identifiers) are not allowed to contain periods. Because we represent types at runtime for marshaling purposes (Sections 5.2.1, 5.4.5, 5.5.4) this label will also denote a global resource that is the type's representation.

By importing worlds, types, and values with `extern` declarations, the compilation unit we write is like a functor whose argument signature is the set of declarations. However, unlike functor arguments these are definite references [54, 129, 130]; they refer to specific resources by label rather than to whatever argument happens to be passed to the functor when it is instantiated.

Having declared the type, we can now import functions that permit access to DOM nodes, for example:

```
<html>
<body>
 <p>Please enter your name:
  <input type="text" name="nom" />
  <input type="submit" value="go" />
 </p>
</body>
</html>
```

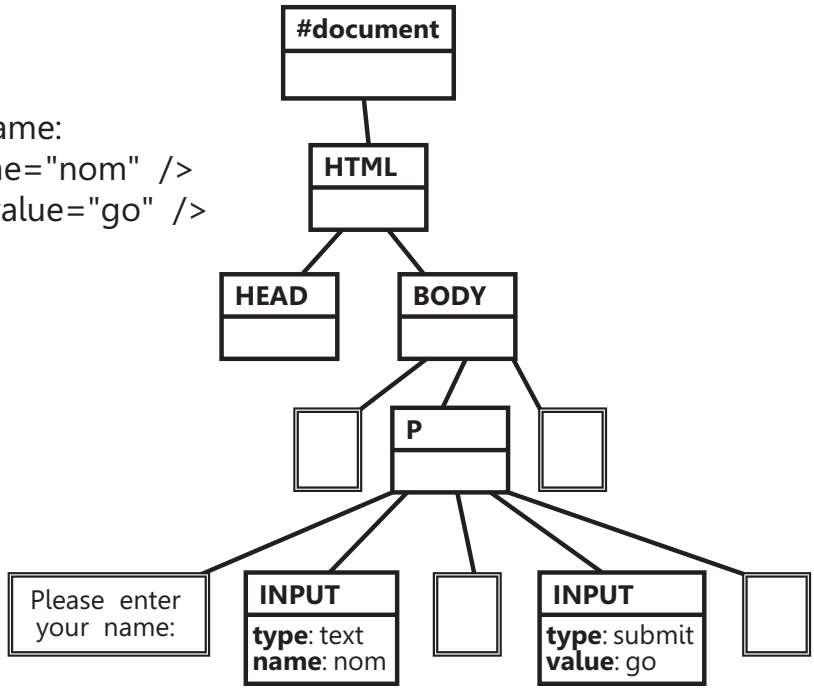

Figure 5.1: An example HTML document and its Document Object Model tree. There are two kinds of nodes, elements and text nodes. Elements may have attributes (such as for the `INPUT` nodes) and children. Text nodes are shown here with a double outline; most consist only of whitespace. The DOM is an abstract representation, correcting for syntactic and semantic errors in the input (such as the missing `<HEAD>` in this document).

```
extern val dom.getbyid :
   string -> dom.node @ home = lc_document_getelementbyid
extern val dom.getstring :
   dom.node * string -> string @ home = lc_domgetobj
extern val dom.setstring :
   dom.node * string * string -> unit @ home = lc_domsetobj
```

Each of these is supported by a small function in the runtime system that implements its behavior, because these are actually member method calls on nodes. The first retrieves a DOM node by its name, given as a string. The `dom.getstring` function allows us to retrieve a property of a node, named by a string and whose value is a string. (Unfortunately, many things in JavaScript are represented by strings. A more structured interface to the DOM would be desirable, but it is outside the scope of this thesis.) For example, we can change the contents of a form field (`<input>` element) named `abox` on the page:

```
val b = dom.getbyid [abox]
do dom.setstring (b, [value], [hello])
```

In order for us to interact with page elements they must already exist in the page; Server 5 provides us with a document that has one element called `page`. We can begin the application by creating elements within the page. A convenient way to do this is to set the `innerHTML` property, which renders HTML into a DOM tree. For example, this program creates a page with an input box and then displays its contents in an alert message:

```
unit
  import "std.mlh"
  import "dom.mlh"

  do dom.setstring
      (dom.getbyid [page],
       [innerHTML],
       [<input id="abox"
         value="hello, world" />])

  val b = dom.getbyid [abox]
  do alert (dom.getstring (b, [value]))

end
```

The header file `"dom.mlh"` contains the `extern` declarations above and others for manipulating the DOM.

Say

To make our application interactive, we can attach handlers to DOM elements that execute code when events occur. For example, we can create a span of text that reacts to

mouse click events:

```
(* ... *)
do dom.setstring
   (dom.getbyid [page],
    [innerHTML],
    [<span onClick="alert('clicked!');">click me</span>])
```

The `onClick` attribute of a DOM element is JavaScript code to be executed when the element is clicked. In this example the JavaScript code calls the familiar `alert` function. We'd prefer to use ML5 to write event handlers, so we can use the `say` construct to produce JavaScript (as a string) from an ML5 expression:

```
(* ... *)
fun handle_click () = alert [clicked!]
do dom.setstring
   (dom.getbyid [page],
    [innerHTML],
    [<span onClick="[say handle_click ()]">\
     click me</span>])
```

This program has the same behavior as the above. The body of the `say` construct can be any ML5 expression (here it is a function call), and the semantics are as follows: At the time the `say` is evaluated, the body is suspended and converted to a JavaScript expression. When that JavaScript expression is evaluated (because it is used as a handler attribute), the suspended ML5 expression is evaluated as a new thread. This allows us to dynamically generate handlers and to modify the behavior of the page at runtime.

Some events have parameters. For example, the `onKeyUp` event for input boxes indicates the key that was pressed. Because the way that events work in JavaScript is quite irregular (Section 5.4.9), we must explicitly specify the event properties that we want and bind them to ML5 variables. For example, we can detect when the enter key is pressed in a form input:

```
do dom.setstring
   (dom.getbyid [page],
    [innerHTML],
    [<input type="text"
      onkeyup="[say { event.keyCode = c }
                 case c of
                   ?\r => alert [pressed enter!]
                 | _ => ()]"
      />])
```

The syntax `?x` is the character constant `x`, and `\r` is the escape sequence for the return character.

Again, it would be nice to provide a more structured interface for interactivity that allows us to use the type system to prevent mistakes. As it stands, the programmer can modify the string that results from a `say` (or simply write his own JavaScript that corrupts the ML5 runtime environment). Given the nature of JavaScript, this would

take a bit of work to accomplish.

The interfaces in the example applications are all built around this way of interacting with the DOM on the client. On the server, most applications interface with a simple database described using the network signature mechanism. However, the purpose of ML5 is not simply to provide glue to resources but to give an expressive language for doing computation with them. In the next section I tour the ML-like features to highlight the differences with Standard ML and ML5, mention how they interact with the modal type system, and to set the stage for a discussion of their implementation.

5.1.4 ML-like features

Datatypes

ML5 supports datatypes in a similar fashion to Standard ML. There are a few minor differences. First, we require that datatypes be uniform [98] and enforce this syntactically. For example, the declaration of the polymorphic list type is

```
infixr ::
datatype a list =
    nil
  | :: of a * list
```

(Also note that ML5 does not use an apostrophe to distinguish type variables, instead using the same identifier conventions as the rest of the language.) Within the declaration of `list`, the identifier `list` refers to the inductive variable of the type being defined, rather than the constructor that will be bound. This means that it is impossible to write non-uniform declarations such as

```
(* SML *)
datatype ('a, 'b) funny =
   A of ((int, bool) funny, 'a * 'a) funny
 | B of ('b, 'a) funny
 | C
```

Such declarations are almost useless because it is not possible to recurse over instances of them due to SML's lack of polymorphic recursion. Ironically, because Standard ML requires equality functions to be generated for any equality type, the compiler must generate a polymorphically recursive function to implement equality for `('a, 'b) funny`! The ML5 internal language does not have any way to represent nonuniform datatypes, nor polymorphic recursion, so we prevent them syntactically. We also do not have equality types of any sort.

Another kind of nonuniformity comes from mutually-recursive datatypes; in ML5 we require that they share the same set of type variables, which are bound at the beginning:

```
datatype (a, b) t1 = X of a | Y of t2
     and        t2 = Z of b | W of t1
```

A datatype may have no arms at all; this is the definition of the standard `void` type ($\bot$):

```
datatype void =
```

In ML5, datatype declarations are transparent, meaning that they do not define new types like Standard ML [137]. Rather, they declare the constructors and bind type names to primitive recursive sum types that "already exist." This means that if the programmer declares the same datatype (up to reordering of constructors) in two places, then those types will be equivalent. It also makes it easier to perform optimizations, since we have more information about the representation when its type is not abstract.

Because datatypes are transparent, constructors can safely be bound in the valid context—they could just as well have been defined anywhere. This is important so that different worlds can share common data structures (and particularly important for types like `option` and `bool`).

The other way that datatypes require special consideration in ML5 is with regard to the mobile judgment. We allow a datatype to be mobile when every datatype in its bundle of mutually-recursive types consists only of arms with mobile types. This will be made precise when we discuss the ML5 mobile judgment in Section 5.3.2.

Extensible types

Datatypes are closed disjoint unions where all of the possibilities (tags) are known at the time of declaration. ML5 also supports extensible types, which are an "open" alternative to datatypes where new tags can be created dynamically. Standard ML has one such type, `exn` (which also happens to be the type of exception tags). In addition to `exn`, ML5 supports the creation of new extensible types. For example, this declaration makes a new extensible type called `exp`:

```
tagtype exp
```

We can then declare new tags for this type:

```
newtag Bool of bool in exp
newtag If of exp * exp * exp in exp
```

They are consumed by pattern matching:

```
fun eval (e : exp) =
    case e of
      If (c, e1, e2) =>
         (case eval c of
            Bool true => eval e1
          | Bool false => eval e2)
    | Bool b => Bool b
```

Interestingly, because extensible types are abstract (as opposed to transparent, like datatypes) any extensible type is mobile. This turns out to be required to make the implementation of exceptions work. The reason that this is safe is that any given constructor (tag) for an arm of an extensible type is not necessarily available at all worlds. For example, given the above declaration of `exp` and `Bool`, the following program is ill-typed:

```
put x = If(Bool true, Bool false, Bool true)
do  from server
    get case x of
          Bool b => b
```

The `put` is allowed because its body has mobile type (`exp`). However, the case analysis is ill-typed because the constructor `Bool` is modally typed at the client, not the server. It is possible, however, to declare all or some of the tags in an extensible type to be valid. For instance, this program is well-typed:

```
tagtype exp
newvtag Bool of bool in exp
newvtag If of exp * exp * exp in exp
newtag  Ref of exp ref in exp

put x = If (Bool true, Ref (ref (Bool false)), Bool false)
do  from server
    get  case x of
           Bool b => 0
         | If (a, b, c) => 1
         | _ => 2
```

The `newvtag` declaration creates a valid tag that can be used in any world. The type that it contains must be mobile. The reason is that the constructor gives us permission to project out the contents of an extensible type at the world where we possess the constructor. If we were allowed to declare the constructor `Ref` to be valid in the example above, then we would be able to pattern match against it on the server and retrieve the reference cell that was allocated at the client. Since reference cells are local resources, this would be unsound. Nonetheless, local resources can be safely injected into extensible types, even from different worlds, because only the world that tagged the value with a modal tag has permission (the tag) to retrieve it.

Extensible types therefore permit more flexibility than datatypes. However, because the set of tags is not known at compile-time, the compiler provides no exhaustiveness checking on patterns and produces less efficient code for generating tags and case-analyzing them. Because of this, extensible types are rarely used (but see Section 5.5.1 for a nice example in the implementation of Server 5)—not nearly as much as features lacking from ML5, like a module system! The only reason that they are supported is that they are an easy generalization of the `exn` type, which is needed for exceptions anyway.

Exceptions

ML5 provides an exception mechanism for non-local control flow just like Standard ML's. An extensible type `exn` with a valid tag `Match` is part of the initial environment. New tags can be created by

```
exception Fail of string
```

which is equivalent to

```
newtag Fail of string in exn
```

(It is also possible to declare valid exceptions with `vexception`.) An exception is thrown by `raise` and caught by `handle`:

```
( raise Fail "oops" )
   handle Fail s => alert [failure: [s]]
        | Match => alert [match?]
```

and an unmatched exception is automatically reraised. The `exn` type, like other extensible types, is mobile. We use this fact during compilation (Section 5.4.2). The only interesting thing about exceptions in ML5 is their interaction with `get`: When we raise an exception in the body of a `get`, the exceptional control flow should propagate back to the calling world. The implementation of this happens in CPS conversion when we translate `get` to `go` (Section 5.4.2).

Functions and pattern matching

ML5 has clausal function declarations and pattern matching similar to Standard ML. There are some extensions and restrictions, mostly syntactic.

Non-clausal patterns are restricted to irrefutable ones. For example, the following is not allowed

```
val (h :: t) = map f l
```

because of the possibility that the right-hand side may be `nil`. (In Standard ML, such patterns can raise the `Bind` exception and have unavoidable unexhaustive match warnings. They also have a suspicious interaction with polymorphic generalization, for example in `val (h :: t) = nil`.)

Because constructor application cannot be curried, application patterns are right-associative:

```
case x of
  SOME SOME y => y
```

For uniformity, case analyses and clausal function declarations can be completely empty (no cases) and thus always raise `Match`. This allows the `abort` construct of lambda calculus for eliminating the $\bot$ type to naturally fall out of a 0-ary case analysis on a datatype with zero arms.

'When' patterns. ML5 also supports a pattern that may perform computation, which is called a *when* pattern. The syntax is

$$\text{patterns} \quad p \quad ::= \quad \ldots \mid (e)\, p$$

where e is an expression. This expression, which must have function type, is applied to the case object and the result is matched against the pattern p. If the expression raises `Match`, then the pattern match also fails. This allows us to implement something like views [139] in a lightweight way; for example, we can pattern match against integers as if they are natural numbers as follows:

```
fun Z 0 = ()
fun S 0 = raise Match
  | S n = n - 1

fun fact (x : int) =
    case x of
       (Z) () => 1
     | (S) n => x * fact n
```

The functions `Z` and `S` are destructors for natural numbers: `Z` matches 0 and `S` matches any non-zero number, with its contents being the predecessor. Note that they raise `Match` in the case they do not match; `Z` by being inexhaustive and `S` explicitly. In the implementation of factorial, we use 'when' patterns to distinguish the two cases.

This form of pattern is particularly useful when patterns are nested and there is no other place to perform computation. (In this example we could have called a function on `x`, for instance, and pattern matched on its result.) One application of this is the "Wizard" interface for abstract datatypes [84], which is used in the ML5/pgh compiler (Section 5.4.1).

First-class continuations

ML5 supports first-class continuations (also sometimes known as `call/cc`) as well. We can obtain the current continuation with `letcc` and activate it with `throw`:

```
val x =
  letcc k
  in
      throw 5 to k;
      6
  end
```

(This code results in `5` being bound to `x`.) Continuation variables are never valid and have type A `cont` where A is the type of value that they expect (`k` has type `int cont` above). The continuation type is not mobile.

Continuations are somewhat esoteric, but find several uses in ML5. First, because a continuation is a natural representation for a thread (a computation that never returns), we use them in the interface to some resources that start threads. For example, the server has a very simple database oriented around string key-value pairs. Its interface is as follows:

```
extern val trivialdb.read : string -> string @ server
extern val trivialdb.update : string * string -> unit @ server
extern val trivialdb.addhook
    : string * unit cont -> unit @ server
```

The functions for reading and updating keys are straightforward. We can also register a hook to be activated (as a new thread) when a specific key changes. We use this in many of the applications to get asynchronous updates of events.

Another use of continuations is for "early exits" from code. This function computes the product of a list of integers but stops if it sees a zero:

```
fun product l =
  letcc ret
  in list-foldl (fn (0, _) => throw 0 to ret
                  | (m, n) => m * n) 1 l
  end
```

In ML, exceptions are typically used for this purpose. This idiom is more direct and more efficient because it does not require the creation of tags or dispatch on them.

Continuations are usually employed in these two stylized ways, but sometimes they are useful in their generality for the creation of user interface prompts. The chat application (Section 6.2) has such an example.

5.1.5 Summary

This concludes the high-level tour of ML5. The language contains other minor features and I have not given an explanation of the constructs that are common with ML. Such a discussion would be better suited to a programming language manual; the purpose of this dissertation is to investigate the type theory and the mechanics of its implementation. Therefore, let us now shift gears and transition to a systematic account of the implementation and the language's definition via elaboration.

5.2 ML5/pgh

ML5/pgh ("ML5 of Pittsburgh") is the compiler for ML5 and—in the absence of a formal semantics for the language—its definition qua reference implementation.

We have already seen several examples, so let us jump straight into the details of the implementation. We'll begin with a description of the compiler's design, particularly with regard to data marshaling, since this will have a pervasive effect on the way we compile programs.

5.2.1 Design concerns

To implement the `get` primitive, which is the centerpiece of the language, we need to be able to transmit data between hosts on the network. Since these hosts are connected via network sockets, which can send only bytes, we need to be able to represent any value as bytes so that it can be transmitted. This process is known as marshaling (it is also sometimes called "serialization" or "pickling") and its inverse is known as unmarshaling.

As discussed in Chapter 2, in ML5 we allow any value to be transmitted between worlds. For example, we can create a local reference cell and wrap it with the `at` modality and it becomes a portable value that can be moved to a remote world. The process

of closure conversion does this whenever it builds an environment containing remote resources, for example. This means that we need to be able to marshal any kind of value.

Marshaling "plain old data"—strings and integers and aggregations of them—is easy. We will be able to marshal code easily as well, because after closure conversion and hoisting, every piece of code can be identified by a global label that can be represented as an integer. The hosts involved in a computation will agree on this set of labels and receive the code for them before the program begins. A problem is posed by polymorphism, however: How do we marshal some value whose type we don't know? For example, consider this version of the polymorphic identity function that prints a message on the server each time it is called:

```
fun (a) id (x : a) =
  let in
     from server get display "called id...\n";
     x
  end
```

(The syntax `(a)` binds a type variable so that we can use it in the ascription for `x`.) After CPS and closure conversion, the value of `x` will need to make a round-trip to the server and back, so we must marshal and unmarshal it. However, this function can be called with any type `a`, so we don't know ahead of time what shape the value of `x` might have and therefore how to marshal it! There are two potential solutions to this problem. The first is to use a uniform representation for values that allows us to programmatically discover their shape at runtime. This essentially requires adding a tag to every value. Many language implementations work this way, because there are other reasons (garbage collection being a common one) that an implementation needs this information at runtime. (The Grid/ML compiler [85] did this for the sake of marshaling, as well.) The other way to do it is to have data (tags) that describe the shape of the values, but to disembody them from the values themselves. Conceptually, these data are the run-time representations of types. For example, we can rewrite the above function to take another parameter:

```
fun (a) id' (tag : a rep, x : a) = (* ... *)
```

The type `a rep` is a singleton type, containing the run-time value that represents the type `a`. If we have this run-time representation around, then it can be an additional input to the marshaling routine and guide its processing of `x`. Decoupling tags from values in this way has a few advantages. First, it constrains our implementation less, because we can use native representations for values and treat the tag data as ancillary. Second, when the type representations are not needed (because we do not attempt to marshal), we can eliminate them from the code. This means that the programmer does not need to pay a performance penalty for the feature when he is not using it.

In the ML5/pgh internals we will perform a type representation transformation (Sections 5.4.5 and 5.4.7) to make sure that whenever we marshal a value, we have a representation of that value's type. Because the modal type system assigns both types and worlds to values, we additionally represent worlds at run-time. This is important because it allows us to specialize the representation of some value given its world. For

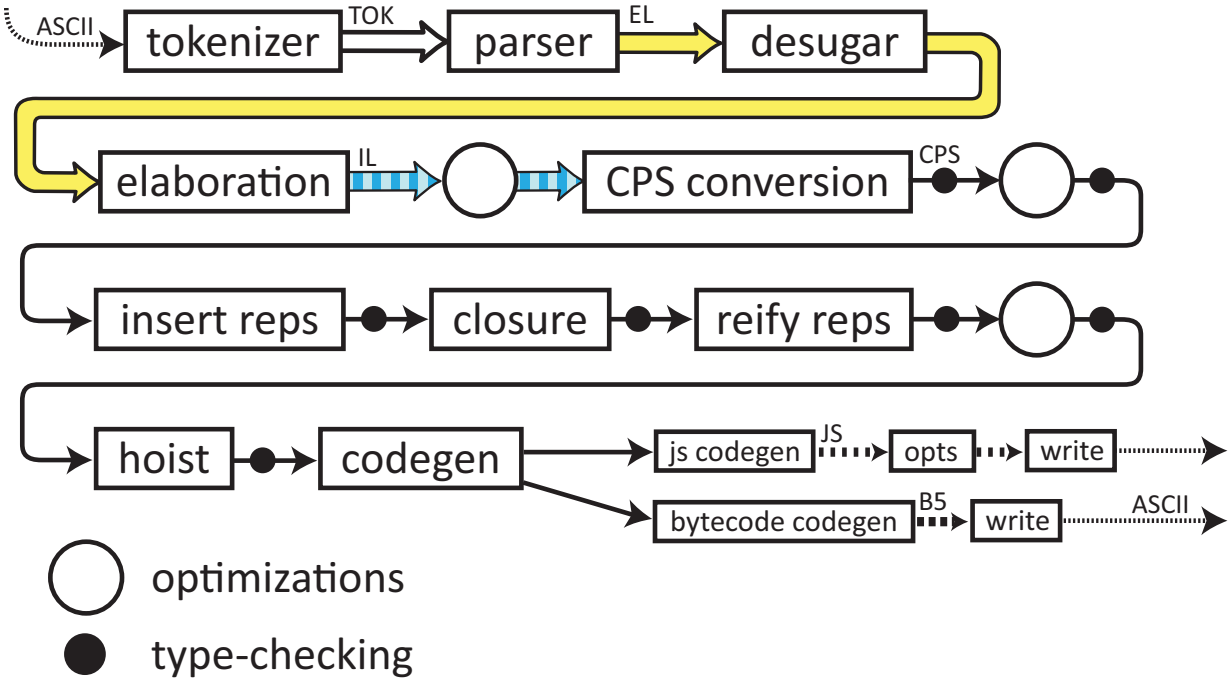

Figure 5.2: The architecture of the ML5/pgh compiler.

example, local resources like DOM handles cannot be easily marshaled, because we do not have control over their representation. Instead, we can do the following. When executing at `home`, we represent values of type `dom.node @ home` as native JavaScript DOM nodes. When executing on another world `server`, we represent a value of type `dom.node @ home` as an integer. This integer is an index into a table of values at `home`. When at `home` we marshal a DOM node value, we insert it in this table and send an integer in its place. When at `home` we unmarshal a `dom.node @ home`, which is represented as an integer, we look it up in the table and return the native pointer. Anywhere else in the network, a DOM node is represented as an integer. (This process, called *desiccation* and *reconstitution*, is detailed in Section 5.5.4.) To use this representation we must know the world of a value as we marshal it; because of world polymorphism we must therefore represent worlds as we do types.

The overall architecture of the compiler appears in Figure 5.2. The front-end (Section 5.3) lexes, parses, and elaborates the source text into the internal language. After a phase of optimization we convert to CPS, where most of the compiler's work is done. We insert the type representations and perform closure conversion, maintaining the invariant that every type and world variable is paired with a representation for it. We then reify these representations into actual data. Hoisting pulls each closed piece of code out to the top-level and gives it a global label. Code generation produces code for each of these labels, depending on what world(s) it needs to be defined for. For each world we output either JavaScript or Server 5 bytecode.

It is worth mentioning a few other principles that guide the implementation. First,

it is a type-directed compiler, meaning that each of these languages (up to code generation) has (possibly implicit) type information associated with it. Second, because this is a prototype implementation not intended for production use, not much concern has been given to the performance of the compiler itself. Instead, an emphasis has been placed on simple and correct code. For example, after every phase in the CPS language, we type-check the entire program again. All transformations are done on functional data structures. Additionally, the interface to the CPS language has been engineered to automatically alpha-vary code to prevent bugs resulting from variable capture.

5.3 Front-end

Now let us discuss the phases of the implementation. The front-end of the compiler is the interface with the user: It reads the source program, type checks it (providing error messages if it is ill-formed), and produces intermediate language code that the rest of the compiler works on.

5.3.1 Parsing

Naturally, the first step is the parser, which reads the input files and produces a data structure representing the external language's abstract syntax. Most parsers are quite boring, specifying the language's grammar in BNF and using a tool like yacc to produce a program implementing the grammar. ML5's parser is somewhat unusual in that it is written with parser combinators [42, 58]. Therefore I will discuss it briefly.

Parser combinators are a way of compositionally hand-writing parsers in functional languages. We begin by defining a type

$$(\alpha, \beta)\ \texttt{parser}$$

which can be thought of as a function that parses some prefix of a β `stream` into an α, returning the remainder of the stream as well. Additionally, the parser may fail. Such parsers may be composed; for example, the sequential parser

$$\texttt{val \&\&} : (\alpha_1, \beta)\ \texttt{parser} \times (\alpha_2, \beta)\ \texttt{parser} \rightarrow (\alpha_1 \times \alpha_2, \beta)\ \texttt{parser}$$

successively applies the two parsers to the same stream and returns a pair of the results. Through careful design of the names of the combinators, the program implementing the parser resembles the grammar of the object language. For example, here is a fragment of the ML5 parser for declarations:

```
fun regulardec G =
 !!(alt[`VAL >> tyvars && (call G pat suchthat irrefutable) &&
          `EQUALS && call G exp
          wth (fn (tv, (pat, (_, e))) => Bind(Val, tv, pat, e)),

        `VAL -- punt "expected val declaration after VAL",
```

```
      'TAGTYPE >> id wth Tagtype,
      'NEWTAG >> expid && opt ('OF >> typ) && 'IN && id
        wth (fn (i,(to,(_,ty))) => Newtag (i, false, to, ty)),

      (* ... *)])
```

The particular meaning of the combinators is not important here; simply observe that each case begins with a keyword token (like `val` or `tagtype`), then the rule in the grammar for that construct, which refers to other syntactic classes as parser functions defined above (or mutually recursive with this one). It is not difficult to give error messages, at least for the grammar of ML5—after seeing a keyword like `val`, if we do not successfully parse a `val` declaration, we can abort and report the current location to the user along with a message. This is the purpose of the second line starting with `'VAL` above.

One nice thing about parsing combinators is that we have the whole power of our general purpose programming language at our disposal. This means that we are not constrained by the particular algorithm that our parser-generator uses. (This is particularly relevant because most real programming languages are not actually context-free.) In the case of ML5, this allows us to properly parse fixity declarations like `infix` by passing along a parsing context to each parsing function. (This is the argument `G` above.) Most ML compilers instead parse a different grammar and resolve fixity after the fact. ML5's nested string constants are implemented with no particular trouble. We even properly parse Standard ML's notoriously difficult nested pattern match [79]:

```
fun hard 0 =
    case x of
       1 => 2
     | 3 => 4
  | hard 5 = 6
```

Therefore, as long as some care is given to error messages, I believe that parser combinators are a good way to implement grammars for ML. Performance is fine, there is no metaprogramming or dependency on other tools, writing the code is a joy, and there is no danger of getting "stuck" being unable to express the desired grammar in a restricted language.

Syntactic sugar

After parsing, there is a small pass to regularize the language by eliminating some syntactic sugar. For example, in the translated code

```
datatype bool = true | false

fun not true : bool = false
  | not false = true
```

we rewrite the declaration of `bool` to take zero type arguments, and to be applied to zero arguments wherever it is used. Now we can treat every datatype declaration uniformly as being polymorphic (perhaps in zero arguments). More importantly, we rewrite the variable patterns in the argument of `not` to be application patterns. This makes pattern compilation much easier because all tag dispatches will be application patterns. We could rewrite these constructors to take "`of unit`" so that every constructor carries a type; instead we internally support a special "`of --`" and a corresponding application pattern to permit more efficient representations of non-carrying constructors.

5.3.2 The internal language

The next step of compilation will be to elaborate the external language program into a more type-theoretic internal language. This includes type and world inference, the implementation of lexical conventions, and the expansion of heavyweight features (*e.g.* datatypes) into primitive type constructors. To explain elaboration we must first present the internal language. We'll begin with a description of its types.

Types and mobility

ML5 has a richer set of types than Lambda 5 did. The major way that these interact with the modal features is via the mobile judgment. The language of types appears in Figure 5.3; these are the *internal language* types that result from elaboration of an ML5 source language. This is the target of type inference, and so polymorphism is explicit. Like Standard ML, ML5 has prenex polymorphism, meaning that type quantifiers all appear at the outside. We maintain this through the internal language by having two levels of type: *polytypes*, which may have quantification, and *monotypes* which do not.

The `at` type, $\mathfrak{S}$ and `addr` types are as before; A `ref` is a reference to a value of type A, and A `array` is an array of A values. An A `cont` is a first-class continuation expecting a value of type A, and α is a type variable. (Primitive types such as `string` and `int` are bound as type variables in the initial context.) The type $\{\ell_1 : A_1, \ldots\}$ is an unordered record with distinct named fields; the empty record is the unit type. The type $[\ell_1 : A_1^?, \ldots]$ is an unordered sum with distinct named arms; the empty sum is the void type. Like in ML, an arm in a sum may optionally carry no value. For example, the type of booleans is $[\mathbf{true} : —, \mathbf{false} : —]$.

A value of type A $\mathtt{tag}_\alpha$ is a dynamically-generated tag for the existential type α. It can be used to tag a value of type A or to extract such a value from the extensible type.

To support mutually recursive functions, we have the individual function type R which is a projection from a bundle of functions $[R_0, \ldots]$. Functions may take multiple arguments.

To support mutually recursive inductive types, we have $\pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_n.A_n)$ which represents $n+1$ mutually-recursive types locally named α_0 to α_n, where the i^{th} one has been selected. All of the bound variables are bound within each of the arms. For example, in the mutually recursive datatype

$$
\begin{array}{rrcl}
\text{worlds} & \mathrm{w} & ::= & \omega \mid \mathbf{w} \\
\text{monotypes} & A & ::= & \alpha \mid \{\ell_1 : A_1, \ldots, \ell_n : A_n\} \mid [\ell_1 : A_1^?, \ldots, \ell_n : A_n^?] \\
 & & \mid & R \mid [R_0, \ldots, R_n] \\
 & & \mid & A\ \mathtt{cont} \mid A\ \mathtt{ref} \mid A\ \mathtt{array} \mid A\ \mathtt{tag}_\alpha \\
 & & \mid & A\ \mathtt{at}\ \mathrm{w} \mid \mathfrak{B}_\omega A \mid \mathrm{w}\ \mathtt{addr} \\
 & & \mid & \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_n.A_n) \\
\text{optional type} & A^? & ::= & — \mid A \\
\text{arrow} & R & ::= & (A_1, \ldots, A_n) \to A \\
\text{polytypes} & P & ::= & \forall\omega_1 \ldots \omega_m.\forall\alpha_1 \ldots \alpha_n.A \\
\text{kinds} & K & ::= & \mathtt{Reg} \mid \mathtt{Ext}
\end{array}
$$

$$\frac{\Gamma(\alpha) = K}{\Gamma \vdash \alpha\ \mathsf{ok}_A} \qquad \frac{}{\Gamma \vdash \mathbf{w}\ \mathsf{ok}_\mathrm{w}} \qquad \frac{\Gamma(\omega) = \ \mathsf{world}}{\Gamma \vdash \omega\ \mathsf{ok}_\mathrm{w}}$$

$$\frac{\Gamma, \omega_1\ \mathsf{world}, \ldots, \omega_m\ \mathsf{world}, \alpha_1 :: \mathtt{Reg}, \ldots, \alpha_n :: \mathtt{Reg} \vdash A\ \mathsf{ok}_A}{\Gamma \vdash \forall\omega_1 \ldots \omega_m.\forall\alpha_1 \ldots \alpha_n.A\ \mathsf{ok}_P}$$

$$\frac{}{\Gamma \vdash —\ \mathsf{ok}_{A^?}} \qquad \frac{\Gamma \vdash A\ \mathsf{ok}_A}{\Gamma \vdash A\ \mathsf{ok}_{A^?}} \qquad \frac{\Gamma(\alpha) = \mathtt{Ext} \quad \Gamma \vdash A\ \mathsf{ok}_A}{\Gamma \vdash A\ \mathtt{tag}_\alpha\ \mathsf{ok}_A}$$

$$\frac{\Gamma \vdash R\ \mathsf{ok}_R}{\Gamma \vdash R\ \mathsf{ok}_A} \qquad \frac{\Gamma \vdash R_0\ \mathsf{ok}_R \quad \ldots \quad \Gamma \vdash R_n\ \mathsf{ok}_R}{\Gamma \vdash [R_0, \ldots, R_n]\ \mathsf{ok}_A}$$

$$\frac{\Gamma \vdash A\ \mathsf{ok}_A}{\Gamma \vdash A\ \mathtt{cont}\ \mathsf{ok}_A} \qquad \frac{\Gamma \vdash A\ \mathsf{ok}_A}{\Gamma \vdash A\ \mathtt{ref}\ \mathsf{ok}_A} \qquad \frac{\Gamma \vdash A\ \mathsf{ok}_A}{\Gamma \vdash A\ \mathtt{array}\ \mathsf{ok}_A}$$

$$\frac{\Gamma \vdash \mathrm{w}\ \mathsf{ok}_\mathrm{w}}{\Gamma \vdash \mathrm{w}\ \mathtt{addr}\ \mathsf{ok}_A} \qquad \frac{\Gamma, \omega\ \mathsf{world} \vdash A\ \mathsf{ok}_A}{\Gamma \vdash \mathfrak{B}_\omega A\ \mathsf{ok}_A} \qquad \frac{\Gamma \vdash A\ \mathsf{ok}_A \quad \Gamma \vdash \mathrm{w}\ \mathsf{ok}_\mathrm{w}}{\Gamma \vdash A\ \mathtt{at}\ \mathrm{w}\ \mathsf{ok}_A}$$

$$\frac{\Gamma \vdash A\ \mathsf{ok}_A \quad \Gamma \vdash A_1\ \mathsf{ok}_A \quad \ldots \quad \Gamma \vdash A_n\ \mathsf{ok}_A}{\Gamma \vdash (A_1, \ldots, A_n) \to A\ \mathsf{ok}_R} \qquad \frac{\begin{array}{l} i \in \{0, \ldots, n\} \\ \Gamma, \alpha_0 :: \mathtt{Reg}, \ldots, \alpha_n :: \mathtt{Reg} \vdash A_0\ \mathsf{ok}_A \\ \vdots \\ \Gamma, \alpha_n :: \mathtt{Reg}, \ldots, \alpha_n :: \mathtt{Reg} \vdash A_n\ \mathsf{ok}_A \end{array}}{\Gamma \vdash \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_n.A_n)\ \mathsf{ok}_A}$$

Figure 5.3: Types of the ML5 internal language. Each syntactic class C has a well-formedness judgment ok_C; the rules are given in this figure because they illustrate the binding structure of types. In the remainder of the discussion, however, we will assume that types are well-formed for brevity.

$$\frac{}{\Gamma, \alpha \text{ mobile} \vdash \alpha \text{ mobile}} \qquad \frac{}{\Gamma \vdash A \,\mathtt{at}\, \mathrm{w} \text{ mobile}}$$

$$\frac{}{\Gamma \vdash \mathfrak{B}_\omega A \text{ mobile}} \qquad \frac{}{\Gamma \vdash \mathrm{w}\ \mathtt{addr} \text{ mobile}}$$

$$\frac{\Gamma \vdash A_1 \text{ mobile} \quad \ldots \quad \Gamma \vdash A_n \text{ mobile}}{\Gamma \vdash \{\ell_1 : A_1, \ldots, \ell_n : A_n\} \text{ mobile}} \qquad \frac{\Gamma \vdash A_i^? \text{ mobile} \quad (\text{where } A_i^? \neq —)}{\Gamma \vdash [\ell_1 : A_1^?, \ldots, \ell_n : A_n^?] \text{ mobile}}$$

$$\frac{\Gamma \vdash A \text{ mobile}}{\Gamma \vdash A\ \mathtt{tag}_\alpha \text{ mobile}}$$

$$\frac{\begin{array}{c}\Gamma, \alpha_0 \text{ mobile}, \ldots, \alpha_n \text{ mobile} \vdash A_0 \text{ mobile} \\ \vdots \\ \Gamma, \alpha_0 \text{ mobile}, \ldots, \alpha_n \text{ mobile} \vdash A_n \text{ mobile}\end{array}}{\Gamma \vdash \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_n.A_n) \text{ mobile}}$$

Figure 5.4: The mobile judgment for ML5. The context Γ here contains assumptions of the form α mobile. This context will include base types like `int` and `string`, as well as any extensible type.

```
datatype top = X of bot | Y
     and bot = Z of top
```

the type `top` is

$$\pi_0(\mu\ \alpha_{\mathtt{top}}.[\mathbf{X} : \alpha_{\mathtt{bot}}, \mathbf{Y} : —],\ \alpha_{\mathtt{bot}}.[\mathbf{Z} : \alpha_{\mathtt{top}}])$$

and the type `bot` is $\pi_1 \ldots$ for the same bundle.

Because of the presence of type variables, the mobile judgment is now made relative to a context, which is a set of type variables assumed to be mobile. Its definition appears in Figure 5.4. Some of the base types such as `int` and `string` are assumed to be mobile in the initial context. The most significant addition to what we had in Lambda 5 is the mobility of inductive types. When checking if an inductive type is mobile, we assume that it (and the other types in its bundle) are mobile. For example, integer lists as produced by the following declaration are mobile:

```
datatype intlist = Nil | Cons of int * intlist
```

However, the types resulting from the following declaration are not mobile:

```
datatype top = X of bot | Y
     and bot = Z of top -> top
```

This is because *all* of the types in the bundle must be mobile for any projection to be mobile, and function types (`top -> top`) are never mobile.

Terms

The IL term language appears in Figure 5.5. As before, we have a syntactic distinction between values and expressions. This is important because of some syntactic value

$$
\begin{array}{rrcl}
\text{vals} & v & ::= & x\langle\vec{A};\vec{\mathrm{w}}\rangle \mid u\langle\vec{A};\vec{\mathrm{w}}\rangle \mid n \mid \texttt{"string"} \\
 & & \mid & \{\!\{\ell_1 = v_1, \ldots, \ell_m = v_m\}\!\} \mid \mathbf{inj}_A^\ell\ v \mid \mathbf{inj}_A^\ell\ \text{—} \\
 & & \mid & \mathbf{held}_{\mathrm{w}}\ v \mid \mathbf{sham}\ \omega.v \mid \mathbf{roll}_A\ v \\
 & & \mid & \mathbf{fns}\ (f_1(x_{11}{:}A_{11}, \ldots, x_{1m}{:}A_{1m}) = M_1, \ldots) \mid \mathbf{fsel}\ v.n \\
\text{exps} & M, N & ::= & \texttt{value}\ v \mid M\ (M_1, \ldots, M_m) \mid \{\ell_1 = M_1, \ldots, \ell_m = M_m\} \\
 & & \mid & \#\ell\ M \mid \texttt{raise}_A\ M \mid M\ \texttt{handle}\ x.N \\
 & & \mid & (M; N) \mid \texttt{let}\ D\ \texttt{in}\ M \mid \texttt{unroll}\ M \\
 & & \mid & \texttt{roll}_A\ M \mid \texttt{say}\ (\ell_1{:}A_1, \ldots, \ell_m{:}A_m)\ M \\
 & & \mid & \texttt{get}[\mathrm{w}; M]\ N \mid \texttt{throw}_A\ M\ \texttt{to}\ N \mid \texttt{letcc}_A\ x\ \texttt{in} N \\
 & & \mid & \texttt{tag}\ M\ \texttt{with}\ N \\
 & & \mid & \texttt{untag}\ M\ \ \texttt{with}\ N\ \texttt{of}(\ \texttt{yes}\ \Rightarrow x.N_1 \mid \texttt{no} \Rightarrow N_2) \\
 & & \mid & \texttt{primapp}\ p\langle\vec{A}\rangle(\vec{M}) \mid \texttt{jointext}\ \vec{M} \\
 & & \mid & \texttt{inj}_A^\ell\ v \mid \texttt{inj}_A^\ell\ \text{—} \\
 & & \mid & \texttt{sumcase}\ M\ \texttt{of}\ (\ell_1 \Rightarrow x.N_1 \mid \ldots \mid \ell_m \Rightarrow x.N_m \mid \_ \Rightarrow N) \\
 & & \mid & \texttt{intcase}\ M\ \texttt{of}\ (n_1 \Rightarrow N_1 \mid \ldots \mid n_m \Rightarrow N_m \mid \_ \Rightarrow N) \\
\text{integers} & n & ::= & 0 \mid 1 \mid -1 \mid \ldots \\
\text{decls} & D & ::= & \texttt{do}\ M \mid \texttt{tagtype}\ \alpha \mid \texttt{newtag}_\alpha\ x\ \texttt{of}\ A \\
 & & \mid & \texttt{val}\ x = M \\
 & & \mid & \texttt{polyval}\ (\vec{\alpha}, \vec{\omega})\ x = v \\
 & & \mid & \texttt{polyput}\ (\vec{\alpha}, \vec{\omega})\ u = v \\
 & & \mid & \texttt{polyleta}\ (\vec{\alpha}, \vec{\omega})\ x = v \\
 & & \mid & \texttt{polyletsham}\ (\vec{\alpha}, \vec{\omega})\ u = v \\
 & & \mid & \texttt{extern world}\ J\ \mathbf{w} \\
 & & \mid & \texttt{extern type}\ \alpha = \ell \\
 & & \mid & \texttt{extern val}\ u \sim \forall\omega.\vec{\omega}.\forall\vec{\alpha}.A = \ell \\
 & & \mid & \texttt{extern val}\ x : \forall\vec{\omega}.\forall\vec{\alpha}.A@\mathrm{w} = \ell \\
\text{worldkinds} & J & &
\end{array}
$$

Figure 5.5: The ML5 internal language terms. We distinguish values and expressions as before, with the boldface version of a construct being the value form. Many constructs take a list of arguments; we use the syntax $\vec{\alpha}$ (for example) to denote a sequence of type variables α.

restrictions (for polymorphism and the body of a **sham** constructor). Many of the constructs are self-explanatory, having appeared before or having direct analogues in ML; I will only explain the ones that are new.

Every variable x (which is bound at a polytype) is applied to a sequence of types and worlds, since terms are assigned monotypes. The same is true for valid variables u. The typing rules appear in Figure 5.6 along with the other rules for values. Most of these are straightforward. There are two forms of injection into sum types, for those constructors that carry a type and those that don't. An inductive type $\pi_i(\mu \ldots)$ can be unrolled one level by selecting the i^{th} component and substituting all of the inductive projections for its free type variables. The **roll** construct is the injection into an inductive type; for it to be well-formed, the value must be of the unrolled type. The value for a bundle of mutually-recursive functions, **fns**, is complicated only because of the mutual recursion and multiple arguments. Each function is bound within the body of all of the functions. The **fsel** value retrieves an individual function from a bundle; this is how we get a value of arrow type that we can call.

The typing rules for expressions appear in Figure 5.7. Most of these are straightforward as well. The rule for `let` uses the typing judgment for declarations,

$$\Gamma \vdash D \overset{\mathrm{w}}{\rightsquigarrow} \Gamma'$$

which means that in the context Γ, the declaration D yields at w the new bindings in Γ'. The rules for declarations are discussed below. The `unroll` expression exposes one level of an inductive type by substituting projections from the bundle for all of the free variables. The `say` construct names the event fields that it expects, and their types, and then supplies a continuation that takes those events as a record. It can only be run at the constant world **home**, which is the client web browser, although this could be relaxed to any world whose worldkind is `javascript` if we supported more than two worlds. The `jointext` construct is primitive string concatenation. There are also primitives for many other operations, such as the allocation of references and arrays, subscripting and comparing strings and other base types, integer math, comparisons, etc.. Their typing rules are not interesting so I do not give them here. A value can be tagged with a compatible tag; the result is some extensible type. (All extensible types are type variables.) To deconstruct a value of extensible type, we test it against a specific tag. If the tag matches, then we retrieve the value embedded within it and proceed along the `yes` branch; if not then we get nothing and proceed along the `no` branch. There are two other kinds of case analysis as well. The `sumcase` construct destructs a labeled n-ary sum, by providing some subset of the labels as distinguished cases and a default in case none of them match. Within each of the distinguished cases a variable is bound to the carried value, unless the label is a non-carrier. We also have `intcase` for efficient dispatch on integers; its rule is similar except that no variable is bound.

Declarations are interesting because they introduce polymorphic and valid bindings. The typing rules for declarations appear in Figure 5.8. The `tagtype` declaration binds a new type variable whose kind is `Ext`. Such variables can be used to make member tags with `newtag`. We have a basic `val` binding that evaluates an expression and binds

$$\frac{\Gamma = \Gamma_1, x{:}\forall\omega_1 \ldots \omega_m.\forall\alpha_1 \ldots \alpha_n.B\,@\,\mathrm{w}, \Gamma_2}{\Gamma \vdash x\langle \vec{A}, \vec{\mathrm{w}}\rangle : [\,{}^{\vec{\mathrm{w}}}/_{\vec{\omega}}][\,{}^{\vec{A}}/_{\vec{\alpha}}]B\,@\,\mathrm{w}} \quad \frac{\Gamma = \Gamma_1, u{\sim}\omega.\forall\omega_1 \ldots \omega_m.\forall\alpha_1 \ldots \alpha_n.B, \Gamma_2}{\Gamma \vdash u\langle \vec{A}, \vec{\mathrm{w}}\rangle : [\,{}^{\mathrm{w}}/_{\omega}][\,{}^{\vec{\mathrm{w}}}/_{\vec{\omega}}][\,{}^{\vec{A}}/_{\vec{\alpha}}]B\,@\,\mathrm{w}}$$

$$\frac{}{\Gamma \vdash n : \mathtt{int}\,@\,\mathrm{w}} \quad \frac{}{\Gamma \vdash s : \mathtt{string}\,@\,\mathrm{w}}$$

$$\frac{\Gamma \vdash v_1 : A_1\,@\,\mathrm{w} \quad \ldots \quad \Gamma \vdash v_m : A_m\,@\,\mathrm{w}}{\Gamma \vdash \{\!\{\ell_1 = v_1, \ldots, \ell_m = v_m\}\!\} : \{\ell_1 : A_1, \ldots \ell_m : A_m\}\,@\,\mathrm{w}}$$

$$\frac{A = [\ell_1 : A_1^?, \ldots, \ell : —, \ldots, \ell_m : A_m^?]}{\Gamma \vdash \mathbf{inj}_A^{\ell}\ — : A\,@\,\mathrm{w}} \quad \frac{\begin{array}{l}\Gamma \vdash v : B\,@\,\mathrm{w}\\ A = [\ell_1 : A_1^?, \ldots, \ell : B, \ldots, \ell_m : A_m^?]\end{array}}{\Gamma \vdash \mathbf{inj}_A^{\ell}\ v : A\,@\,\mathrm{w}}$$

$$\frac{\Gamma \vdash v : A\,@\,\mathrm{w}}{\Gamma \vdash \mathbf{held}_{\mathrm{w}}\ v : A\ \mathtt{at}\ \mathrm{w}\,@\,\mathrm{w}'} \quad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A\,@\,\omega}{\Gamma \vdash \mathbf{sham}\ \omega.v : \beth_{\omega} A\,@\,\mathrm{w}}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : B\,@\,\mathrm{w}\\ 0 \le i \le m\\ A = \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_m.A_m)\\ B = [\,{}^{\pi_0(\mu\ \alpha_0.A_0,\ldots,\alpha_m.A_m)}/_{\alpha_0}] \ldots [\,{}^{\pi_m(\mu\ \alpha_0.A_0,\ldots,\alpha_m.A_m)}/_{\alpha_m}]A_i\end{array}}{\Gamma \vdash \mathbf{roll}_A\ v : A\,@\,\mathrm{w}}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, f_1{:}R_1, \ldots, f_n{:}R_n\\ \Gamma', x_{11}{:}A_{11}, \ldots, x_{1m_1}{:}A_{1m_1} \vdash M_1 : B_1\,@\,\mathrm{w} \qquad R_1 = (A_{11}, \ldots, A_{1m_1}) \to B_1\\ \qquad\qquad \vdots\\ \Gamma', x_{n1}{:}A_{n1}, \ldots, x_{nm_n}{:}A_{nm_n} \vdash M_n : B_n\,@\,\mathrm{w} \qquad R_n = (A_{n1}, \ldots, A_{nm_n}) \to B_n\end{array}}{\Gamma \vdash \mathbf{fns}\ (f_1(x_{11}{:}A_{11}, \ldots, x_{1m_1}{:}A_{1m_1}) = M_1, \ldots) : [R_1, \ldots, R_n]\,@\,\mathrm{w}}$$

$$\frac{\Gamma \vdash v : [R_0, \ldots, R_k]\,@\,\mathrm{w} \quad R_n = (A_1, \ldots, A_m) \to B}{\Gamma \vdash \mathbf{fsel}\ v.n : (A_1, \ldots, A_m) \to B\,@\,\mathrm{w}}$$

Figure 5.6: Typing rules for ML5 internal language values. Each value is assigned a monotype and world with the judgment $\Gamma \vdash v : A\,@\,\mathrm{w}$.

$$\frac{\Gamma \vdash v : A@\mathrm{w}}{\Gamma \vdash \texttt{value}\ v : A@\mathrm{w}} \qquad \frac{\Gamma \vdash M : (A_1, \ldots, A_m) \to B@\mathrm{w} \quad \Gamma \vdash N_1 : A_1@\mathrm{w} \quad \cdots \quad \Gamma \vdash N_1 : A_1@\mathrm{w}}{\Gamma \vdash M\ (N_1, \ldots, N_m) : B@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : \{\ell_1 : A_1, \ldots, \ell : A, \ldots, \ell_m : A_m\}@\mathrm{w}}{\Gamma \vdash \#\ell\ M : A@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : \texttt{exn}@\mathrm{w}}{\Gamma \vdash \texttt{raise}_A\ M : A@\mathrm{w}} \qquad \frac{\Gamma \vdash M : A@\mathrm{w} \quad \Gamma, x{:}\texttt{exn}@\mathrm{w} \vdash N : A@\mathrm{w}}{\Gamma \vdash M\ \texttt{handle}\ x.N : A@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : B@\mathrm{w} \quad \Gamma \vdash N : A@\mathrm{w}}{\Gamma \vdash (M; N) : A@\mathrm{w}} \qquad \frac{\Gamma \vdash D \overset{\mathrm{w}}{\rightsquigarrow} \Gamma' \quad \Gamma, \Gamma' \vdash M : C@\mathrm{w}}{\Gamma \vdash \texttt{let}\ D\ \texttt{in}\ M : C@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_m.A_m)@\mathrm{w}}{\Gamma \vdash \texttt{unroll}\ M : [{}^{\pi_0(\mu\ \alpha_0.A_0,\ldots,\alpha_m.A_m)}/_{\alpha_0}] \ldots [{}^{\pi_m(\mu\ \alpha_0.A_0,\ldots,\alpha_m.A_m)}/_{\alpha_m}]A_i@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : \{1 : A_1, \ldots, m : A_m\}\ \texttt{cont@}\mathbf{home}}{\Gamma \vdash \texttt{say}\ (\ell_1{:}A_1, \ldots, \ell_m{:}A_m)\ M : \texttt{string@}\mathbf{home}}$$

$$\frac{\Gamma \vdash M : \mathrm{w}'\ \texttt{addr@w} \quad \Gamma \vdash N : A@\mathrm{w} \quad \Gamma \vdash A\ \mathsf{mobile}}{\Gamma \vdash \texttt{get}[\mathrm{w}'; M]\ N : A@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : B@\mathrm{w} \quad \Gamma \vdash N : B\ \texttt{cont@w}}{\Gamma \vdash \texttt{throw}_A\ M\ \texttt{to}\ N : A@\mathrm{w}} \qquad \frac{\Gamma, x{:}A\ \texttt{cont@w} \vdash N : A@\mathrm{w}}{\Gamma \vdash \texttt{letcc}_A\ x\ \texttt{in}\ N : A@\mathrm{w}}$$

$$\frac{\Gamma \vdash M_1 : \texttt{string@w} \quad \cdots \quad \Gamma \vdash M_n : \texttt{string@w}}{\Gamma \vdash \texttt{jointext}\ M_1 \ldots M_n : \texttt{string@w}} \qquad \frac{\Gamma \vdash N : A\ \texttt{tag}_\alpha@\mathrm{w} \quad \Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \texttt{tag}\ M\ \texttt{with}\ N : \alpha@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : \alpha@\mathrm{w} \quad \Gamma \vdash N : A\ \texttt{tag}_\alpha@\mathrm{w} \quad \Gamma, x{:}A@\mathrm{w} \vdash N_1 : C@\mathrm{w} \quad \Gamma \vdash N_2 : C@\mathrm{w}}{\Gamma \vdash \texttt{untag}\ M\ \ \texttt{with}\ N\ \texttt{of}\ \ \texttt{yes}\ \Rightarrow x.N_1 \mid \texttt{no} \Rightarrow N_2 : C@\mathrm{w}}$$

$$\frac{\begin{array}{lll} \Gamma \vdash M : [\ell_1 : A_1^?, \ldots, \ell_n : A_n^?]@\mathrm{w} & \Gamma \vdash N_i : C@\mathrm{w} & \text{when } A_i^? = — \\ m \leq n \qquad \Gamma \vdash N : C@\mathrm{w} & \Gamma, x{:}A_i^?@\mathrm{w} \vdash N_i : C@\mathrm{w} & \text{when } A_i^? \neq — \end{array}}{\Gamma \vdash \texttt{sumcase}\ M\ \texttt{of}\ (\ell_1 \Rightarrow x.N_1 \mid \ldots \mid \ell_m \Rightarrow x.N_m \mid \_ \Rightarrow N) : C@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : \texttt{int@w} \quad \Gamma \vdash N_i : C@\mathrm{w} \quad \Gamma \vdash N : C@\mathrm{w}}{\Gamma \vdash \texttt{intcase}\ M\ \texttt{of}\ (n_1 \Rightarrow N_1 \mid \ldots \mid n_m \Rightarrow N_m \mid \_ \Rightarrow N) : C@\mathrm{w}}$$

Figure 5.7: Typing rules for ML5 internal language expressions. Expressions are assigned a type and world with the judgment $\Gamma \vdash M : A@\mathrm{w}$. I omit the rules for record, `roll`, and `inj` expressions because they are the same as their value counterparts. There are many rules for `primapp` depending on the primitive operation being applied; they are all straightforward and omitted here for brevity.

the resulting value. There is also a `polyval` binding that is restricted to values and that produces a polymorphic binding. The familiar `put`, `leta`, and `letsham` constructs come in this polymorphic form. (When we `put` an expression in the external language, we first evaluate the expression with `val` and then bind the result with no polymorphic types or worlds using a degenerate `polyval`.) The `extern world` declaration produces no bindings. This is because it declares the existence of a world constant, not a variable. Within this abstract presentation of the IL, world constants are drawn from an unspecified set of constants $\mathbf{w}$, just as integers n are drawn from the set of integers and string constants are drawn from the set of strings. However, the set of constants is less obvious than the integers (except that we know this set contains the initial world **home**). Therefore we ask the programmer to declare these constants. After checking that the programmer does not use any constants that were undeclared, we preserve the set of world constants just so that we can know which world constants to generate code for (the programmer may never otherwise mention them). They will later be hoisted out of the program to determine this set; we could just as well have done this during elaboration. The `extern type` declaration does bind a variable, as do the modal and valid versions of `extern world`.

5.3.3 Elaboration

Elaboration is the process of transforming the external language (EL) abstract syntax into the internal language (IL), possibly rejecting the program with an error message if it is ill-formed. Traditionally there are two kinds of elaboration: a declarative semantics that relates EL programs to IL programs nondeterministically, and an implementation of those semantics as a syntax-directed transformation with reasonable algorithmic behavior. I do not give a complete elaboration semantics in either form, but show some of the interesting declarative rules and discuss how they are implemented algorithmically.

For the sake of this presentation, I will assume simplified versions of the external language constructs and only discuss the interesting ones. For example, functions will take a single argument with no pattern matching. I will ignore non-carrying datatype and extensible type constructors. I will leave out explicit type variables from `val` and `fun` declarations, since they are not necessary there.

Elaboration is based on two judgments: one for elaborating EL expressions into IL expressions and values, and one for elaborating EL declarations into IL declarations and new context entries. The first is written

$$\Gamma \vdash E \dashrightarrow M : A@\mathrm{w}$$

where M is the resulting IL expression from evaluating E and $A@\mathrm{w}$ is its IL type and world. For declarations, we have

$$\Gamma \vdash L \dashrightarrow^{\mathrm{w}}_{D} \Gamma' \mid D_1, \ldots, D_n$$

where Γ' is the new elaboration context entries produced from elaborating the IL declaration L at the world w and D_1 through D_n are the IL declarations produced.

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{do}\ M \overset{\mathrm{w}}{\rightsquigarrow} \cdot}$$

$$\frac{}{\Gamma \vdash \mathtt{tagtype}\ \alpha \overset{\mathrm{w}}{\rightsquigarrow} \alpha{::}\mathtt{Ext}} \qquad \frac{}{\Gamma \vdash \mathtt{newtag}_\alpha\ x\ \mathtt{of}\ A \overset{\mathrm{w}}{\rightsquigarrow} x{:}A\ \mathtt{tag}_\alpha@\mathrm{w}}$$

$$\frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathtt{val}\ x = M \overset{\mathrm{w}}{\rightsquigarrow} x{:}A@\mathrm{w}} \qquad \frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathtt{Reg} \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathtt{polyval}\ (\vec{\alpha}, \vec{\omega})\ x = v \overset{\mathrm{w}}{\rightsquigarrow} x{:}\forall\vec{\omega}.\forall\vec{\alpha}.A@\mathrm{w}}$$

$$\frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathtt{Reg} \vdash A\ \mathsf{mobile} \quad \Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathtt{Reg} \vdash v : A@\mathrm{w}}{\Gamma \vdash \mathtt{polyput}\ (\vec{\alpha}, \vec{\omega})\ u = v \overset{\mathrm{w}}{\rightsquigarrow} u{\sim}\forall\vec{\omega}.\forall\vec{\alpha}.A}$$

$$\frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathtt{Reg} \vdash v : A\ \mathsf{at}\ \mathrm{w}'@\mathrm{w}}{\Gamma \vdash \mathtt{polyleta}\ (\vec{\alpha}, \vec{\omega})\ x = v \overset{\mathrm{w}}{\rightsquigarrow} x{:}\forall\vec{\omega}.\forall\vec{\alpha}.A@\mathrm{w}'}$$

$$\frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathtt{Reg} \vdash v : \mathfrak{B}_\omega A@\mathrm{w}}{\Gamma \vdash \mathtt{polyletsham}\ (\vec{\alpha}, \vec{\omega})\ u = v \overset{\mathrm{w}}{\rightsquigarrow} u{\sim}\omega.\forall\vec{\omega}.\forall\vec{\alpha}.A}$$

$$\frac{}{\Gamma \vdash \mathtt{extern}\ \mathtt{world}\ J\ \mathbf{w}' \overset{\mathrm{w}}{\rightsquigarrow} \cdot} \qquad \frac{}{\Gamma \vdash \mathtt{extern}\ \mathtt{type}\ \alpha = \ell \overset{\mathrm{w}}{\rightsquigarrow} \alpha{::}\mathtt{Reg}}$$

$$\frac{}{\Gamma \vdash \mathtt{extern}\ \mathtt{val}\ u \sim \omega.\forall\vec{\omega}.\forall\vec{\alpha}.A = \ell \overset{\mathrm{w}}{\rightsquigarrow} u{\sim}\omega.\forall\vec{\omega}.\forall\vec{\alpha}.A}$$

$$\frac{}{\Gamma \vdash \mathtt{extern}\ \mathtt{val}\ x : \forall\vec{\omega}.\forall\vec{\alpha}.A@\mathrm{w}' = \ell \overset{\mathrm{w}}{\rightsquigarrow} x{:}\forall\vec{\omega}.\forall\vec{\alpha}.A@\mathrm{w}'}$$

Figure 5.8: Typing rules for ML5 internal language declarations. Declarations are checked with the judgment $\Gamma \vdash D \overset{\mathrm{w}}{\rightsquigarrow} \Gamma'$, indicating that they may be evaluated at the world w and produce the new bindings Γ'.

In both cases, the context Γ is an *elaboration context*, which contains IL hypotheses and other information we need to perform the translation. This includes information about the constructor status of identifiers, and abbreviations for type identifiers that we expand when we encounter them. I will mention these as they are needed.

Let's begin with the case for EL expression identifiers. We have two rules that might apply:

$$\frac{\Gamma(x_{\texttt{id}}) = \forall\omega_1 \ldots \omega_m.\forall\alpha_1 \ldots \alpha_n.B\,@\,\mathrm{w}}{\Gamma \vdash \texttt{id} \dashrightarrow \texttt{value}\ x_{\texttt{id}}\langle A_1, \ldots A_n, \mathrm{w}_1, \ldots, \mathrm{w}_m\rangle : [{}^{\vec{\mathrm{w}}}/_{\vec{\omega}}][{}^{\vec{A}}/_{\vec{\alpha}}]B\,@\,\mathrm{w}}$$

$$\frac{\Gamma(u_{\texttt{id}}) = \omega.\forall\omega_1 \ldots \omega_m.\forall\alpha_1 \ldots \alpha_n.B}{\Gamma \vdash \texttt{id} \dashrightarrow \texttt{value}\ u_{\texttt{id}}\langle A_1, \ldots A_n, \mathrm{w}_1, \ldots, \mathrm{w}_m\rangle : [{}^{\mathrm{w}}/_{\omega}][{}^{\vec{\mathrm{w}}}/_{\vec{\omega}}][{}^{\vec{A}}/_{\vec{\alpha}}]B\,@\,\mathrm{w}}$$

The external language does not have different syntax for modal and valid variables, so an identifier `id` could be either one. The convention $x_{\texttt{id}}$ gives us the IL modal variable corresponding to an identifier; if it is bound in the context then this is a modal variable. Since variable bindings are given polytypes, we nondeterministically apply it to any $\vec{A}$ and $\vec{\mathrm{w}}$. (In the implementation, these types will actually be determined by type inference.) The convention $u_{\texttt{id}}$ similarly gives us the IL valid variable; if that is bound, then we apply it to the polymorphic type and world arguments, and instantiate the world variable at the current world. Context lookup is arranged such that if the programmer shadows an identifier standing for one sort of variable with an identifier standing for the other, the shadowed variable is not found by the $\Gamma(x)$ operation. (Otherwise, both rules might apply.)

We can insert a variable binding with the `val` declaration, for example:

$$\frac{\Gamma \vdash E \dashrightarrow M : A\,@\,\mathrm{w}}{\Gamma \vdash \texttt{val id} = E \dashrightarrow^{\mathrm{w}}_{D} x_{\texttt{id}}{:}A\,@\,\mathrm{w} \mid \texttt{val}\ x_{\texttt{id}} = M}$$

We produce an IL `val` binding along with the new context entry for it. (Here A is actually the polytype $\forall.\forall.A$; that is, with no type or world variables quantified. These empty quantifiers are omitted for syntactic brevity.) In the case that M is an expression, this is our only choice. However, if M is a value then we can choose to make the binding polymorphic or polymorphic and valid.

$$\frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathsf{Reg} \vdash E \dashrightarrow \texttt{value}\ v : A\,@\,\mathrm{w}'}{\Gamma \vdash \texttt{val id} = E \dashrightarrow^{\mathrm{w}}_{D} x_{\texttt{id}}{:}\forall\vec{\alpha}.\forall\vec{\omega}.A\,@\,\mathrm{w}' \mid \texttt{polyleta}\ (\vec{\alpha}, \vec{\omega})\ x_{\texttt{id}} = \mathbf{held}_{\mathrm{w}'}\ v}$$

In this first case, we choose some world and type variables to make the value polymorphic in, and a world w' where it will be typed. The world is unconstrained; this allows us to (for example) declare functions "`@ server`" and "`@ home`" at the top-level in our program without first traveling there. To accomplish the binding at this possibly remote world, we introduce the `at` modality with **held** and immediately eliminate it with `polyleta`.

We can also make a binding that is valid:

$$\frac{\Gamma, \omega\ \mathsf{world}, \vec{\alpha}{::}\mathsf{Reg}, \vec{\omega}\ \mathsf{world} \vdash E \dashrightarrow \texttt{value}\ v : A\,@\,\omega}{\Gamma \vdash \texttt{val id} = E \dashrightarrow^{\mathrm{w}}_{D} u_{\texttt{id}}{\sim}\omega.\forall\vec{\alpha}.\forall\vec{\omega}.A \mid \texttt{polyletsham}\ (\vec{\alpha}, \vec{\omega})\ u_{\texttt{id}} = \mathbf{sham}\ \omega.v}$$

$$\frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathsf{Reg} \vdash E \dashrightarrow \mathtt{value}\ v : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}{\Gamma \vdash \mathtt{leta}\ \mathtt{id} = E \dashrightarrow^{\mathrm{w}}_{D} x_{\mathtt{id}}{:}\forall\vec{\alpha}.\forall\vec{\omega}.A@\mathrm{w}' \mid \mathtt{polyleta}\ (\vec{\alpha},\vec{\omega})\ x_{\mathtt{id}} = v}$$

$$\frac{\Gamma \vdash E \dashrightarrow M : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}{\Gamma \vdash \mathtt{leta}\ \mathtt{id} = E \dashrightarrow^{\mathrm{w}}_{D} y{:}A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}, x_{\mathtt{id}}{:}A@\mathrm{w}' \mid \mathtt{val}\ y = M,\ \ \mathtt{polyleta}\ ()\ x_{\mathtt{id}} = y\langle\rangle}$$

$$\frac{\Gamma, \vec{\omega}\ \mathsf{world}, \vec{\alpha}{::}\mathsf{Reg} \vdash E \dashrightarrow \mathtt{value}\ v : \text{ß}_{\omega}A@\mathrm{w}}{\Gamma \vdash \mathtt{letsham}\ \mathtt{id} = E \dashrightarrow^{\mathrm{w}}_{D} u_{\mathtt{id}}{\sim}\omega\forall\vec{\alpha}.\forall\vec{\omega}.A \mid \mathtt{polyletsham}\ (\vec{\alpha},\vec{\omega})\ u_{\mathtt{id}} = v}$$

$$\frac{\Gamma \vdash E \dashrightarrow M : \text{ß}_{\omega}A@\mathrm{w}}{\Gamma \vdash \mathtt{letsham}\ \mathtt{id} = E \dashrightarrow^{\mathrm{w}}_{D} y{:}\text{ß}_{\omega}A@\mathrm{w}, u_{\mathtt{id}}{\sim}\omega.A \mid \mathtt{val}\ y = M,\ \ \mathtt{polyletsham}\ ()\ u_{\mathtt{id}} = y\langle\rangle}$$

$$\frac{\Gamma \vdash E \dashrightarrow \mathtt{value}\ v : A@\mathrm{w}'}{\Gamma \vdash \mathtt{hold}\ E \dashrightarrow \mathbf{held}_{\mathrm{w}'}\ v : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}}$$

$$\frac{\Gamma \vdash E \dashrightarrow M : A@\mathrm{w}}{\Gamma \vdash \mathtt{hold}\ E \dashrightarrow \mathtt{let}\ \mathtt{val}\ y = M\ \mathtt{in}\ \mathbf{held}_{\mathrm{w}}\ y\langle\rangle : A\ \mathtt{at}\ \mathrm{w}@\mathrm{w}}$$

$$\frac{\Gamma, \omega_{\mathtt{id}}\ \mathsf{world} \vdash E \dashrightarrow \mathtt{value}\ v : A@\omega}{\Gamma \vdash \mathtt{sham}\ \mathtt{id}.\ E \dashrightarrow \mathbf{sham}\ \omega_{\mathtt{id}}.v : \text{ß}_{\omega_{\mathtt{id}}}A@\mathrm{w}}$$

Figure 5.9: Elaboration of the `at` and ß modalities. Each binding has a polymorphic and monomorphic version, depending on whether the body is a value or not. In the monomorphic version, we sequence the evaluation of the expression with `val` and then make a degenerate polymorphic binding that quantifies over no type or world variables. For `hold`, we might be writing the value **held** (which allows its body to be at another world) or the expression `hold` that introduces the `at` modality locally. The `sham` constructor, on the other hand, requires its body to be a value always.

In this case, we choose polymorphic type and world variables, and a hypothetical world ω at which to check the value. If it is well-typed there, we wrap it in the ß_{ω} modality with **sham** and immediately eliminate it with `polyletsham` to create a polymorphic valid binding.

Although they are rarely needed (because of the powerful `val` declaration), we also give the programmer access to the `at` and ß modalities directly. The rules for these appear in Figure 5.9.

Functions

We treat functions as a form of `val` declaration in order to use the same mechanism for polymorphic generalization and validity. The only complication is therefore mutual recursion. We expand the bundle

```
fun f1(x) = E1
and (* ... *)
and fm(x) = Em
```
to
```
val bundle = (fns f1(x) = E1
              and (* ... *)
              and fm(x) = Em)
val f1 = fsel bundle.0
(* ... *)
val fm = fsel bundle.m
```
and then elaborate that. (This requires adding a syntax for mutually recursive function values and projection to the EL abstract syntax, but not to its concrete syntax.) Because a bundle of functions is a value, and selecting a function from a bundle is a value, these may all be generalized and/or made valid (if possible).

The IL has both the mutually recursive **fns** construct and the **fsel** value so elaboration is totally straightforward; I do not give the rules.

Types

Elaborating types is simple. The EL supports a type abbreviation mechanism,
```
type (a, b) t = a * b * int
```
For simplicity, the IL does not have such a construct; we expand them during elaboration. We therefore have another form of hypothesis that can appear in the elaboration context:

$$\texttt{id} = \lambda\vec{\alpha}.A$$

where `id` is an external language identifier, $\vec{\alpha}$ are its type arguments, and A is an internal language type. We expand such abbreviations eagerly when we encounter them.

The rules for type elaboration are given by the judgment

$$\Gamma \vdash T \texttt{ --}\!\!\rightarrow_T A$$

For example, tuples are expanded as records with fields labeled $1 \ldots n$ as in SML:

$$\frac{\Gamma \vdash T_1 \texttt{ --}\!\!\rightarrow_T A_1 \quad \cdots \quad \Gamma \vdash T_n \texttt{ --}\!\!\rightarrow_T A_n}{\Gamma \vdash T_1 * \cdots * T_n \texttt{ --}\!\!\rightarrow_T \{1 : A_1, \ldots, n : A_n\}}$$

Identifiers are elaborated to type variables and type applications are expanded:

$$\frac{\Gamma(\alpha_{\texttt{id}}) :: \mathcal{K}}{\Gamma \vdash \texttt{id} \texttt{ --}\!\!\rightarrow_T \alpha_{\texttt{id}}}$$

$$\frac{\Gamma(id) = \lambda\alpha_1, \ldots, \alpha_n.B\Gamma \vdash T_1 \texttt{ --}\!\!\rightarrow_T A_1 \quad \cdots \quad \Gamma \vdash T_n \texttt{ --}\!\!\rightarrow_T A_n}{\Gamma \vdash (T_1, \ldots, T_n)\ \texttt{id} \texttt{ --}\!\!\rightarrow_T [{}^{A_1}\!/_{\alpha_1}] \cdots [{}^{A_n}\!/_{\alpha_n}]B}$$

The phase after parsing that eliminated syntactic sugar ensured that all defined types appear as applications (possibly to zero arguments).

Datatypes

Datatypes are an amalgam of several type-theoretic constructs. Their elaboration is therefore somewhat intricate. For the sake of this presentation I will give the elaboration rule for a datatype declaration consisting of exactly two types with one type parameter and three total constructors, *i.e.*

```
datatype (a) t1 = AA of t11 | BB of t12
     and     t2 = CC of t21
```

The result of elaboration is two new type constructors `t1` and `t2` that take a single argument, and valid constructors `AA`, `BB`, and `CC` that produce them. Constructors have no special status in the internal language because we simply use labels (derived from the names of the constructors) to distinguish the branches of the sum. However, during elaboration, constructors are special because they can be used in pattern matching and their application is treated as a value. Therefore, in the elaboration context we have hypotheses of the form

$$u_{\mathtt{id}}\ \mathsf{ctor}_{\ell}$$

to inform us that the value variable $u_{\mathtt{id}}$ is a constructor associated with the label ℓ. We consider such hypotheses to be shadowed by a binding for $u_{\mathtt{id}}$ or $x_{\mathtt{id}}$, so that identifiers can lose constructor status if they are rebound. Given this, the elaboration rule for the limited datatype form above is

$$\frac{\begin{array}{c}\Gamma, \alpha_{\mathtt{a}}{::}\mathtt{Reg}, \alpha_{\mathtt{t1}}{::}\mathtt{Reg}, \alpha_{\mathtt{t2}}{::}\mathtt{Reg} \vdash T_{11} \dashrightarrow_T A_{11} \\ \Gamma, \alpha_{\mathtt{a}}{::}\mathtt{Reg}, \alpha_{\mathtt{t1}}{::}\mathtt{Reg}, \alpha_{\mathtt{t2}}{::}\mathtt{Reg} \vdash T_{12} \dashrightarrow_T A_{12} \\ \Gamma, \alpha_{\mathtt{a}}{::}\mathtt{Reg}, \alpha_{\mathtt{t1}}{::}\mathtt{Reg}, \alpha_{\mathtt{t2}}{::}\mathtt{Reg} \vdash T_{21} \dashrightarrow_T A_{21}\end{array}}{\begin{array}{l}\Gamma \vdash \begin{array}{rcl}\mathtt{datatype}\ (\mathtt{a})\ \mathtt{t1} & = & \mathtt{AA}\ \mathtt{of}\ T_{11} \mid \mathtt{BB}\ \mathtt{of}\ T_{12} \\ \mathtt{and}\ \ \ \mathtt{t2} & = & \mathtt{CC}\ \mathtt{of}\ T_{21}\end{array} \dashrightarrow^{\mathrm{w}}_D \\ \quad \left|\begin{array}{l}\mathtt{t1} = \lambda\alpha_{\mathtt{a}}.\pi_0(\mu\ \alpha_0.[\ell_{\mathtt{AA}} : A_{11}, \ell_{\mathtt{BB}} : A_{12}], \alpha_1.[\ell_{\mathtt{CC}} : A_{21}]), \\ \mathtt{t2} = \lambda\alpha_{\mathtt{a}}.\pi_1(\mu\ \alpha_0.[\ell_{\mathtt{AA}} : A_{11}, \ell_{\mathtt{BB}} : A_{12}], \alpha_1.[\ell_{\mathtt{CC}} : A_{21}]), \\ u_{\mathtt{AA}}{\sim}\forall\alpha_{\mathtt{a}}(A_{11}) \to \pi_0(\mu\ \alpha_0.[\ell_{\mathtt{AA}} : A_{11}, \ell_{\mathtt{BB}} : A_{12}], \alpha_1.[\ell_{\mathtt{CC}} : A_{21}]), \\ u_{\mathtt{AA}}\ \mathsf{ctor}_{\ell_{\mathtt{AA}}}, \\ u_{\mathtt{BB}}{\sim}\forall\alpha_{\mathtt{a}}(A_{12}) \to \pi_0(\mu\ \alpha_0.[\ell_{\mathtt{AA}} : A_{11}, \ell_{\mathtt{BB}} : A_{12}], \alpha_1.[\ell_{\mathtt{CC}} : A_{21}]), \\ u_{\mathtt{BB}}\ \mathsf{ctor}_{\ell_{\mathtt{BB}}}, \\ u_{\mathtt{CC}}{\sim}\forall\alpha_{\mathtt{a}}(A_{21}) \to \pi_1(\mu\ \alpha_0.[\ell_{\mathtt{AA}} : A_{11}, \ell_{\mathtt{BB}} : A_{12}], \alpha_1.[\ell_{\mathtt{CC}} : A_{21}]), \\ u_{\mathtt{CC}}\ \mathsf{ctor}_{\ell_{\mathtt{CC}}}\end{array}\right. \\ \mathtt{polyletsham}\ u_{\mathtt{AA}} = \mathbf{sham}(\mathbf{fsel}\ (\mathbf{fns}(f(x) = \mathbf{roll}\_(\mathbf{inj}^{\ell_{\mathtt{AA}}}_{\_}\ x))).0) \\ \mathtt{polyletsham}\ u_{\mathtt{BB}} = \mathbf{sham}(\mathbf{fsel}\ (\mathbf{fns}(f(x) = \mathbf{roll}\_(\mathbf{inj}^{\ell_{\mathtt{BB}}}_{\_}\ x))).0) \\ \mathtt{polyletsham}\ u_{\mathtt{CC}} = \mathbf{sham}(\mathbf{fsel}\ (\mathbf{fns}(f(x) = \mathbf{roll}\_(\mathbf{inj}^{\ell_{\mathtt{BB}}}_{\_}\ x))).0)\end{array}}$$

A datatype is elaborated into a μ type whose bodies are labeled sums. Each of the carried types T_{11}, T_{12}, T_{21} can mention the explicitly quantified type $\alpha_{\mathtt{a}}$, and the names of each of the datatypes in the bundles ($\alpha_{\mathtt{t1}}$, $\alpha_{\mathtt{t2}}$). Because of the syntactic enforcement of uniformity, these types do not appear in application positions. After the declaration, however, `t1` and `t2` are bound as lambdas that take $\alpha_{\mathtt{a}}$ as an argument and expand to

the μ. The IL does not have a type definition mechanism so this is accomplished using the same mechanism we use for EL type abbreviations.

Each constructor is validly bound at the arrow type for injecting into the datatype; this is so that they can be used as if regular functions. We additionally note that they are constructors, however, so that we can pattern match against them and treat their applications as valuable. These bindings are supported by IL declarations that declare the constructor functions, using the $\Box$ modality and immediately eliminating it, as usual. (I have elided the type annotations on the roll and inj here.)

Extensible types

Though syntactically similar, extensible types are elaborated in a completely different way. Most of the EL constructs have direct IL analogues. For example, the elaboration of the declaration of a new extensible type is

$$\frac{}{\Gamma \vdash \texttt{tagtype id} \dashrightarrow^{\mathrm{w}}_{D} \alpha_{\texttt{id}}\texttt{::Ext} \mid \texttt{tagtype}\ \alpha_{\texttt{id}}}$$

and of a new tag

$$\frac{\Gamma(\alpha_{\texttt{t}}) :: \texttt{Ext} \quad \Gamma \vdash T \dashrightarrow_T A}{\begin{array}{l}\Gamma \vdash \texttt{newtag g of}\ T\ \texttt{in t} \dashrightarrow^{\mathrm{w}}_{D} \\ \begin{array}{l|l} x{:}A\ \mathsf{tag}_{\alpha_{\texttt{t}}}\texttt{@w}, & \texttt{newtag}_{\alpha_{\texttt{t}}}\ x\ \texttt{of}\ A, \\ x_{\texttt{g}}{:}(A) \to \alpha_{\texttt{t}}, & \texttt{val}\ x_{\texttt{g}} = \mathbf{fsel}(\mathbf{fns}(f(y) = \texttt{tag}\ y\ \texttt{with}\ x).0) \\ x_{\texttt{g}}\ \mathsf{tagger}_{x} & \end{array}\end{array}}$$

The hypothesis $x_{\texttt{g}}\ \mathsf{tagger}_{x}$ is like the ctor hypothesis for datatype constructors. It enables us to recognize the identifier $x_{\texttt{g}}$ as an extensible type constructor and associates it with the tag value x used for matching against it.

We also have valid tags, which are a bit more interesting:

$$\frac{\Gamma(\alpha_{\texttt{t}}) :: \texttt{Ext} \quad \Gamma \vdash T \dashrightarrow_T A \quad A\ \mathsf{mobile}}{\begin{array}{l}\Gamma \vdash \texttt{newvtag g of}\ T\ \texttt{in t} \dashrightarrow^{\mathrm{w}}_{D} \\ \begin{array}{l|l} x{:}A\ \mathsf{tag}_{\alpha_{\texttt{t}}}\texttt{@w}, & \texttt{newtag}_{\alpha_{\texttt{t}}}\ x\ \texttt{of}\ A, \\ u{\sim}A\ \mathsf{tag}_{\alpha_{\texttt{t}}} & \texttt{put}\ u = x, \\ u_{\texttt{g}}{\sim}(A) \to \alpha_{\texttt{t}}, & \texttt{polyletsham}\ ()\ u_{\texttt{g}} = \mathbf{sham}(\mathbf{fsel}(\mathbf{fns}(f(y) = \texttt{tag}\ y\ \texttt{with}\ u).0)) \\ u_{\texttt{g}}\ \mathsf{tagger}_{u} & \end{array}\end{array}}$$

Here, after generating the tag we make it valid with `put`. (This requires that the type carried by the tag is mobile, which we check.) We can then declare the constructor function to be valid using the standard idiom of introducing the $\Box$ modality and immediately eliminating it.

Case analysis on extensible types expands to iterated use of the `untag` construct in

the IL. For example, the rule for elaborating a case analysis on extensible types is

$$\frac{\begin{array}{l}\Gamma \vdash E \dashrightarrow M : \alpha @ \mathrm{w} \\ \Gamma(\mathtt{T}) = \mathsf{tagger}_x \\ \Gamma, x_{\mathtt{id}}{:}x \vdash E_1 \dashrightarrow N_1 : C @ \mathrm{w} \\ \Gamma \vdash E_2 \dashrightarrow N_2 : C @ \mathrm{w}\end{array}}{\begin{array}{llllll} & \mathtt{case}\ \ \mathtt{E}\ \ \mathtt{of} & & \mathtt{untag}\ M & \mathtt{with}\ v\ \mathtt{of} & \\ \Gamma \vdash & \quad \mathtt{T}\ \ \mathtt{id} \Rightarrow & E_1 \dashrightarrow & & \mathtt{yes} \Rightarrow x_{\mathtt{id}}.N_1 & : C @ \mathrm{w} \\ & \mid \_ \Rightarrow & E_2 & \mid & \mathtt{no} \Rightarrow N_2 & \end{array}}$$

where we also have a similar rule for when `T` is a valid tag.

Network signatures

The `extern` family of declarations are elaborated easily, since the IL has corresponding constructs. For a value import:

$$\frac{\Gamma, \alpha_{\mathtt{a}_1}{::}\mathtt{Reg}, \ldots, \alpha\mathtt{a}_n{::}\mathtt{Reg} \vdash T \dashrightarrow_T A \quad \Gamma \vdash W \dashrightarrow_W \mathrm{w}}{\begin{array}{l}\Gamma \vdash \mathtt{extern\ val}\ (\mathtt{a}_1, \ldots, \mathtt{a}_n)\ \mathtt{id} : T @ W = \mathtt{id}' \dashrightarrow_D^{\mathrm{w}'} \\ x_{\mathtt{id}}{:}\forall \alpha_{\mathtt{a}_1}, \ldots, \alpha\mathtt{a}_n.A @ \mathrm{w} \mid \mathtt{extern\ val}\ x_{\mathtt{id}}{:}\forall \alpha_{\mathtt{a}_1}, \ldots, \alpha\mathtt{a}_n.A @ \mathrm{w} = \mathtt{id}'\end{array}}$$

$$\frac{\Gamma, \omega\ \mathsf{world}, \alpha_{\mathtt{a}_1}{::}\mathtt{Reg}, \ldots, \alpha\mathtt{a}_n{::}\mathtt{Reg} \vdash T \dashrightarrow_T A}{\begin{array}{l}\Gamma \vdash \mathtt{extern\ val}\ (\mathtt{a}_1, \ldots, \mathtt{a}_n)\ \mathtt{id} \sim (\mathtt{wid}.T) = \mathtt{id}' \dashrightarrow_D^{\mathrm{w}} \\ u_{\mathtt{id}}{:}\omega.\forall \alpha_{\mathtt{a}_1}, \ldots, \alpha\mathtt{a}_n.A \mid \mathtt{extern\ val}\ u_{\mathtt{id}} {\sim} \omega.\forall \alpha_{\mathtt{a}_1}, \ldots, \alpha\mathtt{a}_n.A = \mathtt{id}'\end{array}}$$

The elaboration of types is similar. We only allow imported types to have kind `Reg`; they cannot be type constructors or extensible types.

$$\frac{}{\Gamma \vdash \mathtt{extern\ type\ tid} = \mathtt{tid}' \dashrightarrow_D^{\mathrm{w}} \alpha_{\mathtt{tid}}{::}\mathtt{Reg} \mid \mathtt{extern\ type}\ \alpha_{\mathtt{tid}} = \ell_{\mathtt{tid}'}}$$

Because we bind a type variable, if the same type label is imported in multiple places those bound type variables will not be equal. This is not desirable because these imports are supposed to be definite references. There are a variety of ways to remedy this, but the right one is to implement a real separate compilation system based on modules [130]. In the current implementation, the programmer can easily avoid this by only importing a given type once, at the top of his program.

Finally, the `extern world` declaration informs us of the existence of the named world constant. We record this fact of the identifier:

$$\frac{}{\Gamma \vdash \mathtt{extern}\ J\ \mathtt{world\ wid} \dashrightarrow_D^{\mathrm{w}} \mathtt{wid}\ \mathsf{constant} \mid \mathtt{extern\ world}\ J\ \mathbf{w}_{\mathtt{wid}}}$$

When elaborating world expressions, we expect an identifier to either be in the context as one of these constants (in which case it is elaborated to $\mathbf{w}_{\mathtt{wid}}$) or a variable (in which case it becomes $\omega_{\mathtt{wid}}$). Recall that `extern world` declarations are only kept in the IL for the purposes of code generation, so that we know the worlds (and their world-kinds) that we must generate code for.

Pattern matching

Pattern matching in ML5/pgh accounts for about one third of the code implementing elaboration, but very little of it is related to the modal features. Therefore, I will only discuss it briefly.

The algorithm, which is based on the one used by TILT [134], works on a generalization of case analysis to a matrix of rows and columns, *i.e.*,

$$\begin{array}{llllll} \texttt{case} & v_1 : A_1 & \ldots & v_n : A_n & \texttt{of} & \\ \quad | & p_{11} & \ldots & p_{1n} & \Rightarrow & E_1 \\ \quad | & \vdots & \ddots & p_{mn} & \Rightarrow & E_m \\ \quad | & \_ & & & \Rightarrow & E_{\texttt{def}} \end{array}$$

The values v_1 through v_n are already elaborated, and we insist that they are values (usually variables) so that we can duplicate or eliminate them as we compile the pattern. Every EL pattern can be straightforwardly converted to one of these extended patterns. There are then two mutually recursive phases of compilation: *clean* and *reduce*.

Clean. Cleaning the pattern means establishing an invariant about each of the patterns in the matrix: At its outer level, a pattern must be an application pattern, a constant pattern (integer or char) or a wildcard. Cleaning therefore eliminates n-tuples (by exploding them into n new columns that match on the components of the tuple); variables and *as* patterns (by binding the variable within the arm and replacing it with a wild pattern or the *as* pattern); *when* patterns (by applying the when expression and matching on the result); and type constraints (by unification). A clean matrix is then subject to one round of reduction.

Reduce. Reducing a pattern makes the matrix smaller through a variety of transformations. For example, a column that consists of only wildcard patterns can be eliminated. A column that consists of application patterns (extensible and datatype constructors), or of constants, is compiled into a primitive case construct (a `sumcase`, a series of `untag`s or an `intcase`), each with a nested pattern match for the rest of the matrix. We take some care to make sure that these nested patterns do not duplicate code, by hoisting out common pattern matches as functions. Heuristics guide when to apply the reductions; some cause less code duplication than others.

This algorithm does not lend itself well to exhaustiveness checking, so we use a different technique to issue warnings. A special effectless primop `CompileWarn` is generated in the IL code right before the compiler inserts the `raise Match` corresponding to an inexhaustive pattern match. This primop contains the name of the source file and the position from which it arose. As the code is compiled and unreachable branches are eliminated, so are the warning markers. When we finally generate code, we emit the warnings if they still exist. This gives particularly good warning messages, which are also precise because they take into account any optimizations that the compiler performs.

Type inference

Finally, the elaboration relation that I have described is highly nondeterministic, requiring that types and worlds be guessed in order to form a derivation. Elaboration is actually implemented using a variant of Hindley-Milner type inference [27, 75]. We add to the language of IL types a type metavariable X and to worlds a world metavariable O. These are implemented as ML reference cells. During elaboration, we generate new metavariables and perform unification to determine their true identities. For example, consider a simplified rule for function application:

$$\frac{\Gamma \vdash E_1 \dashrightarrow M_1 : (A) \to B \mathbin{@} \mathrm{w} \quad \Gamma \vdash E_2 \dashrightarrow M_2 : A \mathbin{@} \mathrm{w}}{\Gamma \vdash E_1\ E_2 \dashrightarrow M_1\ (M_2) : B \mathbin{@} \mathrm{w}}$$

(It is actually more complicated, because the application of a constructor to a value is a value, etc.) To type-check, it requires us to guess some combination of the types A and B and the world w. To perform type inference, we do something like the following:

elab (`App`(E_1, E_2)) =
    `let` X_B = new-evar() `in`
    `let` $M_1 : A_1 \mathbin{@} \mathrm{w}_1$ = elab E_1 `in`
    `let` $M_2 : A_2 \mathbin{@} \mathrm{w}_2$ = elab E_2 `in`
    `unify-type` A_1 $((A_2) \to X_B)$ `in`
    `unify-world` w_1 w_2 `in`
    $(M_1\ M_2) : X_B \mathbin{@} \mathrm{w}_1$

We know that whatever M_1's type is, it must be a function, so we unify it against a function type. This allows us to name the return type, even if it has not yet been determined. We also unify the worlds of the function and its argument; unification is much simpler here because worlds are not structured.

The interesting part of type inference is the inference of polymorphism and validity. To elaborate a declaration `val x = E`, we elaborate E at an existential world O, in general yielding $M : A \mathbin{@} O$. If M is an expression, we must use the monomorphic IL `val` rule, so we unify O with the current world and are done.

If M it is `value` v, we might make a polymorphic binding. For type polymorphism, we inspect the type A to see if it has any metavariables in it whose identity has not yet been determined ("free" metavariables). If these metavariables do not appear anywhere in the elaboration context, then they will always be free because there will be no way to unify with them again. We therefore instantiate each one with a new type variable α and $\forall$-quantify over these variables.

World polymorphism is similar: We inspect A looking for world metavariables. If any exist and are not bound in the context or are the metavariable O, then we instantiate and $\forall$-quantify those as well.

The reason that we avoid the world O is that it is outside the scope of the $\forall$ quantifiers, since it is part of the judgment, not the type. However, we have a way to generalize it as well. If O is still free and does not appear in the context, then we use the `polyletsham` elaboration rule to produce a valid, world- and type-polymorphic binding. If O is determined or escapes into the context, then we use the `polyleta` elabo-

ration rule to produce a world- and type-polymorphic binding at that possibly remote world.

At the end of elaboration, some metavariables may have not been determined. If this is the case, we set type metavariables to `unit` and world metavariables to **home**.

One complication of type inference is that we have a distinguished class of types: those types that are mobile. For example, if we elaborate the program

```
fun f y =
  let put z = y
  in 0
  end
```

we need to know that the type of `y` is mobile, but we don't know what it is. We handle this by postponing the mobility check (if necessary) until the end of elaboration. In the above program, the declaration of `f` cannot be polymorphically generalized, because the metavariable will be considered "in the context" as it waits for the mobility check at the end of elaboration.[1] However, the program

```
fun g y =
  let put z = hold y
  in 0
  end
```

does produce a polymorphic binding for `g`, because the type of `hold y` is X `at w`, which is mobile for any X.

5.3.4 Optimization

After the program is elaborated, there is a simple optimization phase to eliminate dead code. This can usually discard much of the standard libraries, which is good because the CPS phases are much slower than the IL phase. Conversion to CPS is next; the CPS language is therefore described in the following section.

5.4 The CPS language

The ML5 CPS language resembles the CPS language we saw in the previous chapter, augmented with the features of the ML5 IL. Since most of the work of the compiler is on the CPS language, there several additional constructs as well. For instance, there are constructs for type and world representations, which we need to perform marshaling at runtime. The syntax for the CPS language is given in Figures 5.10, 5.11, and 5.12.

Types. The types that have changed are as follows. Functions have been replaced by continuations, which do not return. However, we still have a bundle of mutually recursive continuations, written $(\vec{A_1}, \ldots, \vec{A_m})$ `conts`. Each may take multiple arguments. We

[1] If we had a kind of bounded quantifier over only mobile types, then we could generalize. Such a construct is complicated and not well-motivated.

$$
\begin{array}{rrcl}
\text{worlds} & \mathrm{w} & ::= & \omega \mid \mathbf{w} \\
\text{types} & A, B & ::= & \alpha \mid \{\ell_1 : A_1, \ldots, \ell_n : A_n\} \mid [\ell_1 : A_1^?, \ldots, \ell_n : A_n^?] \\
 & & \mid & \vec{A}\ \texttt{cont} \mid (\vec{A_1}, \ldots, \vec{A_m})\ \texttt{conts} \\
 & & \mid & A\ \texttt{ref} \mid A\ \texttt{array} \mid A\ \texttt{tag} \\
 & & \mid & A\ \texttt{at}\ \mathrm{w} \mid \mathfrak{B}_\omega A \mid \mathrm{w}\ \texttt{addr} \\
 & & \mid & \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_n.A_n) \\
 & & \mid & \exists \alpha.\vec{A} \mid \forall \langle \vec{\omega}; \vec{\alpha}; \vec{A} \rangle.B \\
 & & \mid & \texttt{exn} \mid \texttt{bytes} \mid A\ \texttt{rep} \mid \mathrm{w}\ \texttt{wrep} \\
\text{optional type} & A^? & ::= & — \mid A
\end{array}
$$

Figure 5.10: The types of the ML5 CPS language.

$$
\begin{array}{rrcl}
\text{exps} & c & ::= & \texttt{call}\ v_f\ \vec{v_a} \mid \texttt{halt} \\
 & & \mid & \texttt{go}[\mathrm{w}, v_a]\ c \mid \texttt{go\_cc}[\mathrm{w}, v_a, v_e, v_f] \mid \texttt{go\_mar}[\mathrm{w}, v_a, v_m] \\
 & & \mid & \texttt{let}\ x = \texttt{marshal}(v, v_r)\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ x = \texttt{primcall}(\ell : \vec{A} \to B)(\vec{v})\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ x = \texttt{native}(p, \vec{v}, \vec{A})\ \texttt{in}\ c \\
 & & \mid & \texttt{let}\ u = \texttt{localhost}()\ \texttt{in}\ c \\
 & & \mid & \texttt{put}\ u = v\ \texttt{in}\ c \mid \texttt{letsham}\ u = v\ \texttt{in}\ c \\
 & & \mid & \texttt{leta}\ x = v\ \texttt{in}\ c \mid \texttt{val}\ x = v\ \texttt{in}\ c \\
 & & \mid & \texttt{unpack}\ \alpha; u_\alpha; \overline{x : A} = v\ \texttt{in}\ c \\
 & & \mid & \texttt{case}\ v\ \texttt{of}\ (\ell_1 \Rightarrow x.c_1 \mid \cdots \ell_n \Rightarrow x.c_n \mid \_ \Rightarrow c) \\
 & & \mid & \texttt{extern}\ \texttt{val}\ x : A@\mathrm{w} = \ell\ \texttt{in}\ c \\
 & & \mid & \texttt{extern}\ \texttt{val}\ u \sim \omega.A = \ell\ \texttt{in}\ c \\
 & & \mid & \texttt{extern}\ \texttt{world}\ J\ \mathbf{w}\ \texttt{in}\ c \\
 & & \mid & \texttt{extern}\ \texttt{type}\ \alpha = \ell\ \texttt{in}\ c \\
 & & \mid & \texttt{extern}\ \texttt{type}\ \alpha = \ell\ \texttt{with}\ \texttt{rep}\ u = \ell'\ \texttt{in}\ c \\
 & & \mid & \texttt{say}\ x = \overline{\ell : A}.v\ \texttt{in}\ c \mid \texttt{say\_cc}\ x = \overline{\ell : A}.v\ \texttt{in}\ c \\
 & & \mid & \texttt{newtag}\ x\ \texttt{of}\ A\ \texttt{in}\ c \\
 & & \mid & \texttt{untag}\ v_o\ \texttt{with}\ v_t\ \texttt{of}\ (\texttt{yes} \Rightarrow x.c \mid \texttt{no} \Rightarrow c') \\
\text{primops} & p & ::= & + \mid - \mid * \mid / \mid < \mid \texttt{stringeq} \mid \texttt{newref} \mid \ldots
\end{array}
$$

Figure 5.11: The continuation expressions of the ML5 CPS language. Again we use $\vec{\alpha}$ or $\overline{x : A}$ to denote a sequence of arbitrary length.

$$
\begin{array}{rrcl}
\text{vals} & v & ::= & x \mid u \\
& & \mid & \mathbf{fns}(f_1(\overline{x_1 : A_1}) = c_1, \ldots, f_n(\overline{x_n : A_n}) = c_n) \\
& & \mid & \mathbf{fsel}\ v.n \mid n \mid \texttt{"string"} \mid \#\ell\ v \\
& & \mid & \{\!\{\ell_1 = v_1, \ldots, \ell_n = v_n\}\!\} \mid \mathbf{held}_{\mathrm{w}}\ v \\
& & \mid & \mathbf{sham}\ \omega.v \mid \mathbf{inj}_\ell^A\ v^? \mid \mathbf{roll}_A\ v \mid \mathbf{unroll}\ v \\
& & \mid & \mathbf{pack}\ A, v_d, \vec{v}\ \mathbf{as}\ \exists\alpha.\vec{B} \\
& & \mid & \mathbf{unpack}\ \alpha; u_\alpha; \overline{x : A} = v\ \texttt{in}\ v' \\
& & \mid & \mathbf{\Lambda}\langle\vec{\omega}; \vec{\alpha}; \overline{x{:}A}\rangle.v \mid v_f\ \langle\vec{\mathrm{w}}; \vec{A}; \vec{v}\rangle \\
& & \mid & \mathbf{tag}\ v\ \mathbf{with}\ v_t \\
& & \mid & \mathbf{leta}\ x = v\ \texttt{in}\ v' \mid \mathbf{letsham}\ u = v\ \texttt{in}\ v' \\
& & \mid & \mathbf{wrepfor}\ \mathrm{w} \mid \mathbf{wrep}\ \mathbf{w} \mid \mathbf{repfor}\ A \\
& & \mid & \mathbf{rep}\ \{\ell_1 : v_1, \ldots, \ell_n : v_n\} \mid \mathbf{rep}\ [\ell_1 : v_1^?, \ldots, \ell_n : v_n^?] \\
& & \mid & \mathbf{rep}\ (\vec{v}\ \texttt{cont}) \mid \mathbf{rep}\ ((\vec{v_1}, \ldots, \vec{v_m})\ \texttt{conts}) \\
& & \mid & \mathbf{rep}\ (v\ \texttt{at}\ v') \mid \mathbf{rep}\ \mathfrak{B}_{(\omega,u)}v \\
& & \mid & \ldots \\
\text{optional vals} & v^? & ::= & v \mid \text{—}
\end{array}
$$

Figure 5.12: The values of the ML5 CPS language. For every syntactic form of type there is a syntactic form of value, its representation. I do not repeat all of these here.

now have a single extensible type, called `exn`; all extensible types in the IL are translated to this one. The type of marshaled data is `bytes`. For closure conversion, we have an existential type. We no longer have a prenex restriction on polymorphism, so we have a first-class polymorphism construct. This $\forall$ type quantifies over a series of world, type, and value variables all at once.

We also have a type of run-time world representations, and a type of type representations. A value of type A `rep` is a representation of the type A; it is a singleton type. Similarly, a value of type w `wrep` is a representation of the world w.

Expressions. The syntax for CPS expressions appears in Figure 5.11 and their typing rules in Figure 5.13. As usual, continuation expressions are typed with a judgment

$$\Gamma \vdash c \star \mathrm{w}$$

which means that c is well-typed to evaluate at the world w. Some constructs (`go`, `say`, and `extern type`) have several versions (which have different typing rules), only one of which will be in use depending on what phase of the compiler we are in. The bare `go` construct takes an address and a continuation for us to execute at the remote world; it does not return. The `go_cc` construct is the same thing post closure-conversion; it takes a closure to execute remotely. The `go_mar` construct takes a marshaled closure. We can marshal any value with `marshal`, as long as we have a representation of its type. (Unmarshaling is implicit in the `go_mar` construct.) For `say`, we have a typing rule similar

to the IL one. After closure conversion, `say_cc` expects a closure-converted continuation, which unfortunately means that the type of a closure-converted continuation is baked into the rule.[2]

The expression `primcall` represents a call to a function imported with the IL `extern val` construct. Such functions cannot be CPS converted, so we need a special way of making direct-style calls to them. The `native` expression is a single primop applied to values; primops include mathematical operations on integers, comparisons, and access to arrays and references. (There are a variety of typing rules, all obvious, so I do not give them here.) As in the IL, `extern world` has no effect on the typing context. The `extern type` expression may additionally declare the label for the type's representation. These are inserted in the first type representation phase (Section 5.4.5). The `newtag` and `untag` expressions are as in the IL except that they now use the single universal type `exn`.

Values. As expected, the typing judgment for values (given in Figure 5.14) is

$$\Gamma \vdash v : A@\mathrm{w}$$

meaning that v has the type A at the world w. It may be surprising, however, what we consider a value in the CPS language. It includes, for instance, **leta** and **letsham**—which bind variables—and other elimination forms such as the application of a $\mathbf{\Lambda}$ to types, worlds, and values. These turn out to be necessary because as we compile the language, things that were once simple values (*e.g.* functions) become more complicated. Because the language has several value restrictions (for the body of **held** and **sham**, for example), we must maintain valueness through these translations. Perhaps a better terminology would be to call continuation expressions "statements" and values "pure expressions." This is because the value restriction is not really a restriction to values but a restriction to non-effectful expressions. Still, this is the terminology we use elsewhere, so we will stick with it here.

We have the usual constructs: variables, continuation bundles, selection from such bundles, integer and string constants, records and projection from them, etc. We have a value **pack** for creating existential types. In addition to the type, this requires a representation of the type. This is so that when we unpack the existential, binding a type variable, we will be able to maintain the invariant that every type variable in scope has a representation also in scope. (The representation must be valid, which we encode by using the $\mathfrak{B}$ modality.) The existential package contains a series of values, which eases closure conversion. Unpacking, which is also a value, binds a type variable and a valid variable to its representation, as well as a variable for each of the packed values.

The value of $\forall$ type is a lambda over a sequence of world, type, and value variables. Its body must be a value, because the application of such a value to actual world, types, and values is also considered a value. The order of the arguments is important, in

[2]It would be nicer to make the `say` construct a primop or feature of the runtime system that we simply import. Unfortunately the code generator must know about it because of problems with the way that events work in JavaScript.

$$\frac{\Gamma \vdash v_f : (A_1, \ldots, A_n)\ \texttt{cont@w} \quad \Gamma \vdash v_1 : A_1\texttt{@w} \quad \cdots \quad \Gamma \vdash v_n : A_n\texttt{@w}}{\Gamma \vdash \texttt{call}\ v_f\ (v_1, \ldots, v_n)\star\mathrm{w}} \qquad \frac{\begin{array}{c}\Gamma \vdash v_a : \mathrm{w}'\ \texttt{addr@w} \\ \Gamma \vdash v_e : A\texttt{@w}' \\ \Gamma \vdash v_f : A\ \texttt{cont@w}'\end{array}}{\Gamma \vdash \texttt{go\_cc}[\mathrm{w}', v_a, v_e, v_f]\star\mathrm{w}}$$

$$\frac{\Gamma \vdash c\star\mathrm{w}' \quad \Gamma \vdash v_a : \mathrm{w}'\ \texttt{addr@w}}{\Gamma \vdash \texttt{go}[\mathrm{w}', v_a]\ c\star\mathrm{w}} \qquad \frac{\Gamma \vdash v_a : \mathrm{w}'\ \texttt{addr@w} \quad \Gamma \vdash v_m : \texttt{bytes@w}}{\Gamma \vdash \texttt{go\_mar}[\mathrm{w}', v_a, v_m]\star\mathrm{w}}$$

$$\frac{\Gamma \vdash v : A\texttt{@w} \quad \Gamma \vdash v_r : A\ \texttt{rep@w} \quad \Gamma, x{:}\texttt{bytes@w} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{let}\ x = \texttt{marshal}(v, v_r)\ \texttt{in}\ c\star\mathrm{w}} \qquad \frac{\Gamma, x{:}A\ \texttt{tag@w} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{newtag}\ x\ \texttt{of}\ A\ \texttt{in}\ c\star\mathrm{w}}$$

$$\frac{\Gamma \vdash v_o : \texttt{exn@w} \quad \Gamma \vdash v_t : A\ \texttt{tag@w} \quad \Gamma, x{:}A\texttt{@w} \vdash c\star\mathrm{w} \quad \Gamma \vdash c'\star\mathrm{w}}{\Gamma \vdash \texttt{untag}\ v_o\ \texttt{with}\ v_t\ \texttt{of yes} \Rightarrow x.c \mid \texttt{no} \Rightarrow c'\star\mathrm{w}}$$

$$\frac{\Gamma \vdash v_1 : A_1\texttt{@w} \quad \cdots \quad \Gamma \vdash v_n : A_n\texttt{@w} \quad \Gamma, x{:}B\texttt{@w} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{let}\ x = \texttt{primcall}(\ell : (A_1, \ldots, A_n) \to B)(v_1, \ldots, v_n)\ \texttt{in}\ c\star\mathrm{w}}$$

$$\frac{\Gamma \vdash v : \exists\alpha.A_1, \ldots, A_n\texttt{@w} \quad \Gamma, \alpha\ \mathsf{type}, u_\alpha{\sim}\alpha\ \texttt{rep}, x_1{:}A_1\texttt{@w}, \ldots, x_n{:}A_n\texttt{@w} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{unpack}\ \alpha; u_\alpha; x_1{:}A_1, \ldots, x_n{:}A_n = v\ \texttt{in}\ c\star\mathrm{w}}$$

$$\frac{\Gamma, x{:}A\texttt{@w}' \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{extern val}\ x : A\texttt{@w}' = \ell\ \texttt{in}\ c\star\mathrm{w}} \qquad \frac{\Gamma, u{\sim}\omega.A \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{extern val}\ u \sim \omega.A = \ell\ \texttt{in}\ c\star\mathrm{w}}$$

$$\frac{\Gamma \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{extern world}\ J\ \mathbf{w}\ \texttt{in}\ c\star\mathrm{w}} \qquad \frac{\Gamma, u{\sim}\mathrm{w}\ \texttt{addr} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{let}\ u = \texttt{localhost}()\ \texttt{in}\ c\star\mathrm{w}}$$

$$\frac{\Gamma, \alpha\ \mathsf{type} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{extern type}\ \alpha = \ell\ \texttt{in}\ c\star\mathrm{w}} \qquad \frac{\Gamma, \alpha\ \mathsf{type}, u{\sim}\alpha\ \texttt{rep} \vdash c\star\mathrm{w}}{\Gamma \vdash \texttt{extern type}\ \alpha = \ell\ \texttt{with rep}\ u = \ell'\ \texttt{in}\ c\star\mathrm{w}}$$

$$\frac{\Gamma \vdash v : (\{1 : A_1, \ldots, m : A_m\})\ \texttt{cont@home} \quad \Gamma, x{:}\texttt{string@home} \vdash c\star\mathbf{home}}{\Gamma \vdash \texttt{say}\ x = (\ell_1{:}A_1, \ldots, \ell_m{:}A_m)\ v\ \texttt{in}\ c\star\mathbf{home}}$$

$$\frac{\Gamma \vdash v : \exists\alpha.\alpha, (\alpha, \{1 : A_1, \ldots, m : A_m\})\ \texttt{cont@home} \quad \Gamma, x{:}\texttt{string@home} \vdash c\star\mathbf{home}}{\Gamma \vdash \texttt{say\_cc}\ x = (\ell_1{:}A_1, \ldots, \ell_m{:}A_m)\ v\ \texttt{in}\ c\star\mathbf{home}}$$

Figure 5.13: The typing rules for ML5 CPS continuation expressions. I omit the typing rules for `put`, `letsham`, `leta`, `val`, `intcase` and `sumcase` because they are essentially the same as they were in the IL.

$$\frac{\Gamma(x) = A@\mathrm{w}}{\Gamma \vdash x : A@\mathrm{w}} \quad \frac{\Gamma(u) = \omega.A}{\Gamma \vdash u : [{}^{\mathrm{w}}/_{\omega}]A@\mathrm{w}}$$

$$\frac{\begin{array}{c}\Gamma', x_{11}{:}A_{11}@\mathrm{w}, \ldots x_{1m}{:}A_{1m}@\mathrm{w} \vdash c_1 \star \mathrm{w} \\ \vdots \\ \Gamma', x_{n1}{:}A_{n1}@\mathrm{w}, \ldots x_{nk}{:}A_{nk}@\mathrm{w} \vdash c_n \star \mathrm{w} \\ \Gamma' = \Gamma, f_1{:}(A_{11}, \ldots, A_{1m})\ \mathtt{cont}@\mathrm{w}, \ldots, f_n{:}(A_{n1}, \ldots, A_{nk})\ \mathtt{cont}\end{array}}{\begin{array}{rcl} f_1(x_{11} : A_{11}, \ldots, x_{1m} : A_{1m}) & = & c_1, \qquad (A_{11}, \ldots, A_{1m}), \\ \Gamma \vdash \mathbf{fns}(\quad \vdots \quad) & : & (\quad \vdots \quad)\ \mathtt{conts}@\mathrm{w} \\ f_n(x_{n1} : A_{n1}, \ldots, x_{nk} : A_{nk}) & = & c_n \qquad (A_{n1}, \ldots, A_{nk}) \end{array}}$$

$$\frac{0 \leq i \leq n \quad \Gamma \vdash v : ((A_{01}, \ldots, A_{0m}), \ldots, (A_{n1}, \ldots, A_{nk}))\ \mathtt{conts}}{\Gamma \vdash \mathbf{fsel}\ v.i : (A_{i1}, \ldots, A_{im})\ \mathtt{cont}@\mathrm{w}}$$

$$\frac{\Gamma \vdash v : A@\mathrm{w}'}{\Gamma \vdash \mathbf{held}_{\mathrm{w}'}\ v : A\ \mathtt{at}\ \mathrm{w}'@\mathrm{w}} \quad \frac{\Gamma, \omega\ \mathsf{world} \vdash v : A@\omega}{\Gamma \vdash \mathbf{sham}\ \omega.v : \mathfrak{B}_{\omega}A@\mathrm{w}} \quad \frac{\Gamma \vdash v : A@\mathrm{w} \quad \Gamma \vdash v_t : A\ \mathtt{tag}@\mathrm{w}}{\Gamma \vdash \mathbf{tag}\ v\ \mathbf{with}\ v_t : \mathtt{exn}@\mathrm{w}}$$

$$\frac{\Gamma \vdash v_d : \mathfrak{B}(A\ \mathtt{rep})@\mathrm{w} \quad \Gamma \vdash v_1 : [{}^{A}/_{\alpha}]B_1@\mathrm{w} \quad \cdots \quad \Gamma \vdash v_n : [{}^{A}/_{\alpha}]B_n@\mathrm{w}}{\Gamma \vdash \mathbf{pack}\ A, v_d, v_1, \ldots, v_n\ \mathbf{as}\ \exists\alpha.(B_1, \ldots, B_n)@\mathrm{w}}$$

$$\frac{\Gamma, \omega_1\ \mathsf{world}, \ldots, \omega_n\ \mathsf{world}, \alpha_1\ \mathsf{type}, \ldots, \alpha_m\ \mathsf{type}, x_1{:}A_1@\mathrm{w}, \ldots, x_k{:}A_k@\mathrm{w} \vdash v : B@\mathrm{w}}{\begin{array}{rl}\Gamma \vdash & \mathbf{\Lambda}\langle\omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_m; x_1{:}A_1, \ldots, x_k{:}A_k\rangle.v \\ & : \forall\langle\omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_m; A_1, \ldots, A_k\rangle.B@\mathrm{w}\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_f : \forall\langle\omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_m; A_1, \ldots, A_k\rangle.B@\mathrm{w} \\ \Gamma, \omega_1\ \mathsf{world}, \ldots, \omega_n\ \mathsf{world}, \alpha_1\ \mathsf{type}, \ldots, \alpha_m\ \mathsf{type} \vdash v_1 : A_1@\mathrm{w} \\ \vdots \\ \Gamma, \omega_1\ \mathsf{world}, \ldots, \omega_n\ \mathsf{world}, \alpha_1\ \mathsf{type}, \ldots, \alpha_m\ \mathsf{type} \vdash v_l : A_l@\mathrm{w}\end{array}}{\Gamma \vdash v_f\ \langle \mathrm{w}_1, \ldots, \mathrm{w}_n; A_1, \ldots, A_m; v_1, \ldots, v_k\rangle : [{}^{\mathrm{w}_1, \cdots, \mathrm{w}_n}/_{\omega_1, \cdots, \mathrm{w}_n}][{}^{A_1, \cdots, A_m}/_{\alpha_1, \cdots, \alpha_m}]B@\mathrm{w}}$$

Figure 5.14: The typing rules for ML5 CPS values. I omit the typing rules for integers, strings, records and record projection, sum injections and case analysis, and roll and unroll for inductive types because they are essentially the same as they were in the IL. I also omit the value versions of `unpack`, `leta`, and `letsham` because they are essentially the same as their expression counterparts. Some of the rules for type and world representation values appear in Figure 5.15.

$$\frac{}{\Gamma \vdash \textbf{wrepfor}\ \mathrm{w}' : \mathrm{w}'\ \texttt{wrep@w}} \quad \frac{}{\Gamma \vdash \textbf{wrep}\ \mathbf{w} : \mathbf{w}\ \texttt{wrep@w}} \quad \frac{}{\Gamma \vdash \textbf{repfor}\ A : A\ \texttt{rep@w}}$$

$$\frac{\Gamma \vdash v : A\ \texttt{rep@w}}{\Gamma \vdash \textbf{rep}\ (v\ \texttt{ref}) : (A\ \texttt{ref})\ \texttt{rep@w}} \quad \frac{}{\Gamma \vdash \textbf{rep}\ \ \texttt{exn} : \texttt{exn}\ \texttt{rep@w}}$$

$$\frac{\Gamma \vdash v_1 : A_1\ \texttt{rep@w} \quad \cdots \quad \Gamma \vdash v_n : A_n\ \texttt{rep@w}}{\Gamma \vdash \textbf{rep}\ (v_1, \ldots, v_n\ \texttt{cont}) : (A_1, \ldots, A_n)\ \texttt{cont}\ \texttt{rep@w}}$$

$$\frac{\Gamma \vdash v_1 : A_1\ \texttt{rep@w} \quad \cdots \quad \Gamma \vdash v_n : A_n\ \texttt{rep@w}}{\Gamma \vdash \textbf{rep}\ \{\ell_1 : v_1, \ldots, \ell_n : v_n\} : \{\ell_1 : A_1, \ldots, \ell_n : A_n\}\ \texttt{rep@w}}$$

$$\frac{\begin{array}{cc} v_i^? = — & \text{when } A_i^? = — \\ \Gamma \vdash v_i^? : A_i^?\ \texttt{rep@w} & \text{when } A_i^? \neq — \end{array}}{\Gamma \vdash \textbf{rep}\ [\ell_1 : v_1^?, \ldots, \ell_n : v_n^?] : [\ell_1 : A_1^?, \ldots, \ell_n : A_n^?]\ \texttt{rep@w}}$$

$$\frac{\Gamma \vdash v : A\ \texttt{rep@w} \quad \Gamma \vdash v' : \mathrm{w}'\ \texttt{wrep@w}}{\Gamma \vdash \textbf{rep}\ (v\ \texttt{at}\ v') : (A\ \texttt{at}\ \mathrm{w}')\ \texttt{rep@w}}$$

$$\frac{\Gamma, \omega\ \mathsf{world}, u{\sim}\omega\ \texttt{wrep} \vdash v : A\ \texttt{rep@w}}{\Gamma \vdash \textbf{rep}\ \mathfrak{B}_{\omega,u} v : (\mathfrak{B}_\omega A)\ \texttt{rep@w}} \quad \frac{\Gamma \vdash v : \mathrm{w}'\ \texttt{wrep@w}}{\Gamma \vdash \textbf{rep}\ (v\ \texttt{addr}) : (\mathrm{w}'\ \texttt{addr})\ \texttt{rep@w}}$$

$$\frac{\begin{array}{l} \Gamma' = \Gamma, \alpha_0\ \mathsf{type}, u_{\alpha_0}{\sim}\alpha_0\ \texttt{rep}, \ldots, \alpha_n\ \mathsf{type}, u_{\alpha_n}{\sim}\alpha_n\ \texttt{rep} \\ \Gamma' \vdash v_0 : A_0\ \texttt{rep@w} \\ \quad \vdots \\ \Gamma' \vdash v_n : A_n\ \texttt{rep@w} \end{array}}{\Gamma \vdash \textbf{rep}\ \pi_i(\mu\ (\alpha_0, u_{\alpha_0}).v_0, \ldots (\alpha_n, u_{\alpha_n}).v_n) : \pi_i(\mu\ \alpha_0.A_0, \ldots \alpha_n.A_n)\ \texttt{rep@w}}$$

$$\frac{\Gamma, \alpha\ \mathsf{type}, u_\alpha{\sim}\alpha\ \texttt{rep} \vdash v_1 : A_1\ \texttt{rep@w} \quad \cdots \quad \Gamma, \alpha\ \mathsf{type}, u_\alpha{\sim}\alpha\ \texttt{rep} \vdash v_1 : A_1\ \texttt{rep@w}}{\Gamma \vdash \textbf{rep}\ (\exists(\alpha, u_\alpha).v_1, \ldots, v_n) : (\exists\alpha.A_1, \ldots, A_n)\ \texttt{rep@w}}$$

$$\frac{\begin{array}{c} \Gamma, \omega_1\ \mathsf{world}, u_{\omega_1}{\sim}\omega_1\ \texttt{wrep}, \ldots, \alpha_1\ \mathsf{type}, u_{\alpha_1}{\sim}\alpha_1\ \texttt{wrep} \vdash v_1 : A_1\ \texttt{rep@w} \\ \vdots \\ \Gamma, \omega_1\ \mathsf{world}, u_{\omega_1}{\sim}\omega_1\ \texttt{wrep}, \ldots, \alpha_1\ \mathsf{type}, u_{\alpha_1}{\sim}\alpha_1\ \texttt{wrep} \vdash v_n : A_n\ \texttt{rep@w} \\ \Gamma, \omega_1\ \mathsf{world}, u_{\omega_1}{\sim}\omega_1\ \texttt{wrep}, \ldots, \alpha_1\ \mathsf{type}, u_{\alpha_1}{\sim}\alpha_1\ \texttt{wrep} \vdash v : B\ \texttt{rep@w} \end{array}}{\Gamma \vdash \textbf{rep}\ (\forall\langle(\omega_1, u_{\omega_1}), \ldots; (\alpha, u_{\alpha_1}), \ldots; v_1, \ldots\rangle.v) : (\forall\langle\omega_1, \ldots; \alpha_1, \ldots; A_1, \ldots\rangle.B)\ \texttt{rep@w}}$$

$$\frac{\Gamma \vdash v : A\ \texttt{rep@w}}{\Gamma \vdash \textbf{rep}\ (v\ \texttt{rep}) : (A\ \texttt{rep})\ \texttt{rep@w}} \quad \frac{\Gamma \vdash v : \mathrm{w}'\ \texttt{wrep@w}}{\Gamma \vdash \textbf{rep}\ (v\ \texttt{wrep}) : (\mathrm{w}'\ \texttt{wrep})\ \texttt{rep@w}}$$

Figure 5.15: The typing rules for ML5 CPS world and type representations, which are values. I omit the straightforward rules for `bytes`, A `array`, A `tag` and `conts`.

that the types of the value arguments may depend on the type arguments and world arguments.

Finally we have world and type representations, whose typing rules appear in Figure 5.15 (though they are values like any other). World representations are simpler. There are two: **wrepfor** w is a stand-in for the representation of the world w. We use this form until we have established an invariant where every world variable ω has associated with it a valid variable of type ω `wrep`, and then replace `wrepfor` ω with that valid variable (Section 5.4.7). A "real" value of type $\mathbf{w}$ `wrep` is **wrep** $\mathbf{w}$ for a concrete world $\mathbf{w}$. These are the representations that are actually passed around at runtime. The stand-in **repfor** A is similar to **wrepfor**. There is one syntactic form of **rep** for each syntactic form of CPS type (except variables). However, where there were type components in the type, there are now value components in the representation. For example, the record type

$$\{\ell_1 : \texttt{exn}, \ell_2 : \alpha\ \texttt{ref}\}$$

is represented by the value

$$\mathbf{rep}\ \{\ell_1 : \mathbf{rep}\ \texttt{exn}, \ell_2 : \mathbf{rep}(u_\alpha\ \texttt{ref})\}$$

if u_α has type α `rep`. Moreover, those syntactic forms of type that bind type or world variables (such as $\mathfrak{B}_\omega A$) must now bind two variables: the type or world variable, and the valid variable standing for its representation. For example, the representation of

$$\mathfrak{B}_\omega(\texttt{exn at}\ \omega)$$

is

$$\mathbf{rep}\ (\mathfrak{B}_{(\omega,u)}\mathbf{rep}((\mathbf{rep}\ \texttt{exn})\ \texttt{at}\ u))$$

where $u{\sim}\omega$ `wrep` is bound within the value's body.

This syntactic redundancy is somewhat annoying, because in the implementation each type constructor must be given twice (as a type and as a value), and any utility code (for pretty-printing, etc.) must also be written twice. Fortunately, the CPS representation that we use in ML5/pgh has a nice interface that removes some of this redundancy. Before we begin describing the translation from IL to CPS and the steps of compilation that take place in the CPS representation, let us discuss its ML implementation.

5.4.1 Return to Oz

In this section I discuss the ML5/pgh implementation of the CPS abstract syntax. Although I think this is a nice technique, it is not a main contribution of the dissertation and not necessary for understanding the remainder of this chapter, and so may be safely skipped.

The standard way of representing an abstract syntax tree in ML is via its algebraic datatype mechanism. For example, a straightforward encoding of the ML5 CPS types in Standard ML would be

```
datatype ctyp =
    At of ctyp * world
  | Cont of ctyp list
  | Conts of ctyp list list
  | Shamrock of var * ctyp
  | Mu of int * (var * ctyp) list
  | (* ... *)
```

This is a very convenient implementation, particularly because of our ability to write functions over datatypes via pattern matching. However, it has a shortcoming, which is that we do not have any control over the representation of the datatype. This can be a problem when we want to do something special with the representation, such as use a more efficient data structure, store additional information, or maintain certain invariants. In these cases what we want from the datatype mechanism is its facility for pattern matching, not its automatic creation of constructors and destructors from the representation it chooses. Unfortunately, ML offers no direct way to decouple the two.[3]

In my undergraduate senior thesis I studied a technique for alternative implementations of datatypes while retaining a form of pattern matching, known as the "Wizard" [84]. The idea is to provide an abstract type (which can be implemented any way we want) and a projection function that produces a single-level datatype that we can pattern match on. For example,

```
type ctyp (* abstract *)
datatype ctypfront =
    At of ctyp * world
  | Cont of ctyp list
  | Conts of ctyp list list
  | Shamrock of var * ctyp
  | Mu of int * (var * ctyp) list
  | (* ... *)
val look : ctyp -> ctypfront
val hide : ctypfront -> ctyp
```

The important thing is that the datatype is only partially revealed; it refers to the abstract type rather than being recursive. To create an element of the abstract type, we repeatedly apply datatype constructors and the `hide` injection. When we want to pattern match on a `ctyp`, we simply call the projection function on the case object:

```
case look t of
  At (t, w) => (* ... *)
| Cont l => (* ... *)
| _ => (* ... *)
```

The biggest drawback is that we can only pattern match one level at a time, because there is no place to put the projection function in a nested pattern match. I found such patterns to be very rare in practice. A refinement of this representation is to make the

[3]Because Standard ML uses the opaque interpretation of datatypes [137], it would not be hard for a future version to support such a mechanism, however.

partially exposed datatype parameterized by a type:

```
type ctyp
datatype 'a front =
    At of 'a * world
  | Cont of 'a list
  | Conts of 'a list list
  | Shamrock of var * 'a
  | Mu of int * (var * 'a) list
  | (* ... *)
val look  : ctyp -> ctyp front
val look2 : ctyp -> ctyp front front
(* ... *)
```

This allows us to provide a projection function that reveals the abstract type to two or more levels, for nested pattern matches. Another useful trick is to provide injective constructors

```
val At'   : ctyp * world -> ctyp
val Cont' : ctyp list -> ctyp
(* ... *)
```

so that we can build elements of the abstract type as if it were a datatype.

The purpose of the original project was to improve the performance of TILT by using a more efficient representation (de Bruijn indices and hash consing) without rewriting the code. Although it was feasible to retrofit a new representation into the compiler with this technique, the performance results were negative. The code was substantially slower when running through the Wizard interface, and—disappointingly—even slower when the representation "optimizations" were turned on.

The ML5/pgh compiler uses a Wizard-like interface for the CPS language, but not for efficiency (indeed, it comes at a performance cost here as well). Instead, we use it to improve the correctness of the code, by providing an interface to the CPS language that helps to avoid alpha-conversion bugs. (Various subtle alpha-conversion mistakes were the predominant source of bugs in the Humlock compiler's CPS language [91] with hand-managed binding.) We do this by making the CPS representation aware of the binding structure of the AST. We provide primitive substitution, renaming, free variable computations and comparisons. Additionally, whenever we look under a binder, we see a new freshly alpha-varied variable that has never been seen before. Programming against this interface thus resembles programming with higher-order abstract syntax (Section 4.7.1) or other high-level representations of binding structure like FreshML's [110].

Both the interface to the CPS language and its implementation are interesting; let us first look at the interface.

CPS interface

The CPS interface looks like the final code example above, with one further trick employed. Because we parameterized the datatype for type fronts, it can equally well be used for the implementation of types and for the implementation of type representation values. It looks like this:

```
datatype ('tbind, 'ctyp, 'wbind, 'world) ctypfront =
    At of 'ctyp * 'world
  | Cont of 'ctyp list
  | TExists of 'tbind * 'ctyp list
  | Shamrock of 'wbind * 'ctyp
  | (* ... *)

type ctyp
val ctyp : ctyp -> (var, ctyp, var, world) ctypfront

datatype ('cexp, 'cval) cexpfront =
    Go of world * 'cval * 'cexp
  | (* ... *)

and      ('cexp, 'cval) cvalfront =
    Held of world * 'cval
  | Rep of (var * var, 'cval, var * var, 'cval) ctypfront
  | (* ... *)

type cval
type cexp
val cval : cval -> (cexp, cval) cvalfront
val cexp : cexp -> (cexp, cval) cexpfront
```

Here the `typfront` type is parameterized over its recursive occurrence `'ctyp` but also the type of type bindings, world bindings, and worlds. When we project from the abstract `ctyp` type, we get a `ctypfront` where the binders are single variables and types are `ctyp`s and worlds are `world`s, as expected. To reuse the datatype (which is much longer than shown here) for type representations, we instead instantiate `'ctyp` and `'world` with the type of values, because the representations recursively contain more representations, not types. Binders are instantiated with pairs of variables: one for the static type or world and one for its representation. This means that a lot of utility code (for example, pretty printing) can be implemented once in a generic way for both types and their representations.

In the interface we also have injective constructors for each arm of the datatypes, functions for computing free variable sets, substitutions, alpha-equivalence respecting total orders, etc. These have the straightforward types.

Again, the nice thing about this interface is that it automatically renames bound variables as we open their binders. This means that every binding we see is globally unique,

so we do not have to worry about name clashes as we do transformations of the code. This simplifies the client's job. Such a representation also discourages the programmer from doing wrong or suspicious things. For example, a naïve program might make a pass over the code to find functions that ought to be inlined, identifying those functions by the variable they are bound to. Making a second pass and expecting to find the functions bound to the same variables is ideologically suspect—names should not matter!—and might even be wrong, if the process of inlining can cause these variables to alpha-vary because of substitution.

CPS implementation

The first implementation of the CPS language was done by hand, writing the renaming and substitution functions over the natural datatype representation. The language is fairly large, so this turned out to be error-prone and difficult to experiment with. This is frustrating because the algorithms are repetitive and structured: each binder or variable occurrence is treated exactly the same way. To avoid writing repetitive code, I reimplemented the CPS language using a second Wizard. The second language, called "AST", is a tiny kernel language with constants, aggregation, binding structure, and nothing else. The language is similar to an untyped LF or Harper's Abstract Binding Trees [49]. The AST language is functorized over the type of variables and leaves:

```
signature ASTARG =
sig
  type var
  val var_cmp : var * var -> order
  val var_vary : var -> var

  type leaf
  val leaf_cmp : leaf * leaf -> order
end
```

For variables, we only need to know how to order them (and therefore test for equality) and to alpha-vary them. For leaves, which can be anything, we only need to know how compare them. Given these types, the interface to the AST is

```
type ast
datatype 'ast front =
    $ of leaf
  | / of 'ast * 'ast
  | \ of var * 'ast
  | V of var

val look   : ast -> ast front
val look2  : ast -> ast front front
(* ... *)
val hide   : ast front -> ast
```

```
val $$ : leaf -> ast
val // : ast * ast -> ast
val \\ : var * ast -> ast
val VV : var -> ast
```

In this language, we can have a leaf, a pair of two ASTs, a variable bound within an AST or a variable occurrence. (The actual implementation also supports a list of ASTs and a list of binders, since these are quite common. I will ignore them for this discussion.) We also get functions for comparing ASTs, substituting for variables, and computing the free variables of an AST. Each time we look under the binder `\`, the variable is renamed to a freshly generated variable that we have never seen before.

The nice thing about this language is that it is quite minimal, so it is possible to easily experiment with different implementations of it. I have developed a few different implementations; the most complex one uses a limited form of explicit substitution to avoid the cost of renaming variables in certain common scenarios. A conformance suite tests that it has the correct behavior.

We use the AST functor to build the implementation of the CPS language. We start by declaring a leaf datatype that has all of the constructor names for our language; these do not carry any values. (For example, there is a constructor `AT_` which we use to encode the `At of ctyp * world` constructor.) We must define a comparison functions on leaves as well. The type of variables is instantiated with a disjoint union of four sorts of variables:

```
datatype var =
    WV of Var.var (* world variable *)
  | TV of Var.var (* type  variable *)
  | MV of Var.var (* modal variable *)
  | VV of Var.var (* valid variable *)
```

Then, we can implement the CPS language by making each of the abstract types `world`, `ctyp`, `cval`, and `cexp` be the `ast` type. The injective constructors encode the CPS language elements into ASTs. For example:

```
fun At'(t, w) = $$AT_ // t // w
fun Shamrock'(wv, t) = $$SHAMROCK_ // (WV wv \\ t)
```

Recall that `$$`, `//` and `\\` (the latter two used with infix notation) are the injective constructors for the AST, where `\\` is the binder. To project out an abstract CPS world, type, value, or expression, we project out the AST term that implements it (usually several levels) and pattern match:

```
fun ctyp (typ : ast) =
  case look2 typ of
    $AT_ / t / w => At(t, w)
  | $SHAMROCK_ / (WV v \ t) => Shamrock(v, t)
  | (* ... *)
  | _ => raise CPS "bad ctyp"
```

Note that we have some costs here. For one, we perform many more run-time tag checks to distinguish the different cases during these pattern matches. Some of these tag

checks result from a loss of static information about what forms a value might take. Each of the syntactic classes is represented uniformly as an AST, and the type system does not tell us that the `AT_` leaf is always followed by a type and a world, for example.[4] These runtime costs are constants, however, and the loss in static checking easy to manage. (Because we create ASTs in a structured way, our destructing pattern matches either work every time or fail immediately.)

As benefits, we get correct substitution functions for all of the constructs of the language, free variable computations, and alpha-conversion respecting comparison functions. Comparisons are used on types during type-checking, naturally, but having a for-free comparison function for expressions and values was quite useful during optimizations (for example, in the hoisting phase we are able to conveniently coalesce code that is identical up to alpha-conversion).

The implementation of the CPS language in terms of AST is only about twice as long as its signature (which contains mainly the datatype declarations), so it is not much code at all. Essentially, it consists of two (injection and projection) explications of the binding structure of the language described by the constructs of AST. Other than this duplication, it is about as terse and direct as could be.

5.4.2 CPS conversion

We first use the CPS implementation by transforming IL code into it, through the process of CPS conversion. CPS conversion is very similar to how we formalized it for MinML5 in Section 4.5. There are only a few things that are conceptually new: exceptions, first-class continuations, polymorphism, and imported values.

Double-barreled CPS

We translate away exceptions during CPS conversion by using "double-barreled CPS." The double barrels are the two continuations passed to each function: the return continuation, used for returning normally, and the exception continuation, used for throwing an exception.

CPS conversion of expressions is given almost as before:

$$\mathsf{convert}\ \Gamma\ \mathcal{K}\ M\ A\ \mathrm{w}$$

where M is the expression to convert, A and w are its IL type and world. The context Γ is a CPS typing context, and $\mathcal{K}$ is a meta-level function indicating how to form the tail of the expression from the new context and CPS value. (As before, $[\![A]\!]_{\mathsf{C}}$ converts an IL type to a CPS type and $[\![v]\!]_{\mathsf{VC}}^{A@\mathrm{w}}$ converts values.) To implement double-barreled CPS, we maintain an additional invariant that Γ contains a modal variable *handler* with type `exn cont` at the current world w. Initially, this continuation simply halts the program.

The CPS conversion of the IL `raise` M expression is the simplest:

[4] A more clever use of the AST would be able to regain some amount of static checking through the use of phantom types, but this code is so simple as to make this overkill.

$[\![\mathbf{fns}(f_1(x_{11}, ldots) = M_1, \ldots)]\!]_{\mathsf{VC}}^{[(A_{11},\ldots,A_{1n})\to B,\ldots]\,@\,\mathrm{w}}$ =
  $\mathbf{fns}(f_1(x_{11}, \ldots, x_r{:}[\![B]\!]_{\mathsf{c}}\ \mathtt{cont}, \mathit{handler}{:}\mathtt{exn}\ \mathtt{cont}) = c_1, \ldots)$
  where c_1 =
    convert $(\Gamma, x_{11}, \ldots, x_r{:}[\![B]\!]_{\mathsf{c}}\ \mathtt{cont})\mathcal{K}\ M_1\ B\ \mathrm{w}$
    where $\mathcal{K}(\Gamma; v)$ = $\mathtt{call}\ x_r\ (v)$

convert $\Gamma\ \mathcal{K}(M\ (N_1, \ldots, N_n))\ B\ \mathrm{w}$ =
  convert $\Gamma\mathcal{K}'\ M\ ((A_1, \ldots, A_n) \to B)\ \mathrm{w}$
  where $\mathcal{K}'(\Gamma; v)$ = convert $\Gamma\ \mathcal{K}'_1\ N_1\ A_1\ \mathrm{w}$
    where $\mathcal{K}'_1(\Gamma; v_1)$ =
      convert $\Gamma\ \mathcal{K}'_2\ N_2\ A_2\ \mathrm{w}$
        $\vdots$
      where $\mathcal{K}'_n(\Gamma, v_n)$ =
      $\mathtt{val}\ x_j = (\mathbf{fn}(x).\mathcal{K}(\Gamma, x{:}[\![B]\!]_{\mathsf{c}}@\mathrm{w}; x))\ \mathtt{in}$
      $\mathtt{call}\ v\ (v_1, \ldots, v_n,\ x_j, \mathit{handler})$

Figure 5.16: Double-barreled CPS conversion of IL functions and applications, which is fairly standard. (For brevity, the bundle of mutually recursive functions is abbreviated to only show the first function and its first argument.)

convert $\Gamma\ \mathcal{K}\ (\mathtt{raise}_A\ M)\ A\ \mathrm{w}$ =
  convert $\Gamma\ \mathcal{K}'\ M\ \mathtt{exn}\ \mathrm{w}$
  where $\mathcal{K}'(\Gamma; v)$ = $\mathtt{call}$ *handler*(v)

We first CPS-convert M, which must be of type `exn`. When we have its value, v, we simply call the current handler on v. Maintaining this handler is the job of the rest of the translation. For example, the IL `handle` expression is converted as follows:

convert $\Gamma\ \mathcal{K}\ (M\ \mathtt{handle}\ x.N)\ A\ \mathrm{w}$ =
  $\mathtt{val}\ x_j = \mathbf{fn}(x_r).\mathcal{K}(\Gamma, x_r{:}[\![A]\!]_{\mathsf{c}}; x_r)\ \ \mathtt{in}$
  $\mathtt{val}$ *handler* = $(\mathbf{fn}(x).$
      convert $(\Gamma, x{:}\mathtt{exn@w})\ \mathcal{K}'\ N\ A\ \mathrm{w}$
        where $\mathcal{K}'(\Gamma; v)$ = $\mathtt{call}\ x_j$ (v)) $\ \mathtt{in}$
  convert $(\Gamma, x_j{:}\mathtt{exn\ \ cont})\ \mathcal{K}''\ M\ A\ \mathrm{w}$
    where $\mathcal{K}''(\Gamma; v)$ = $\mathtt{call}\ x_j$ (v)

The function x_j is the join point for the two possible ways of returning from this expression: either directly or via the handler. ($\mathbf{fn}\ (\vec{x}).c$ is shorthand for a bundle containing a single continuation, which is selected out.) It is simply the reification of the input continuation $\mathcal{K}$. We then shadow the continuation variable *handler* with the new handler. It executes N (with the old handler in scope) and returns to the join point. With this defined, we convert M, which might call our new handler or return normally to the join point.

To maintain the handler dynamically through the call stack, we pass it as an additional argument to every function along with the return continuation (Figure 5.16). The interesting part of the implementation of exceptions is their interaction with `get`. Recall that `get` is translated into two `go`s; one to go to the remote world and one to return. To implement exceptions, we need to make sure that we establish a handler at the remote world that re-raises the exception at the source world. Let's assume that `get` takes an address as a value v_a to shorten the presentation; then its translation is

$$
\begin{array}{l}
\textsf{convert}\ \Gamma\ \mathcal{K}\ (\texttt{get}[\mathrm{w}', v_a]\ N)\ A\ \mathrm{w} = \\
\quad \texttt{val}\ x_{old} = \textit{handler}\ \texttt{in} \\
\quad \texttt{val}\ x_r = \texttt{localhost()}\ \texttt{in} \\
\quad \texttt{put}\ u_r = x_r\ \texttt{in} \\
\quad \texttt{go}[\mathrm{w}'; v_a] \\
\quad\quad \texttt{val}\ \textit{handler} = (\textbf{fn}\ (x_e). \\
\quad\quad\quad\quad\quad \texttt{put}\ u_e = x_e\ \texttt{in} \\
\quad\quad\quad\quad\quad \texttt{go}[\mathrm{w}; u_r] \\
\quad\quad\quad\quad\quad\quad \texttt{call}\ x_{old}\ (u_r))\ \texttt{in} \\
\quad\quad \textsf{convert}\ (\Gamma, x_r{:}\mathrm{w}\ \texttt{addr@w}, u_r{\sim}\mathrm{w}\ \texttt{addr}, x_{old}{:}\texttt{exn cont@w})\ \mathcal{K}'\ \mathrm{N}\ A\ \mathrm{w}' \\
\quad\quad \text{where}\ \mathcal{K}'(\Gamma, v) = \texttt{put}\ u = v\ \texttt{in} \\
\quad\quad\quad\quad\quad\quad \texttt{go}[\mathrm{w}; u_r] \\
\quad\quad\quad\quad\quad\quad\quad \texttt{val}\ \textit{handler} = x_{old}\ \texttt{in} \\
\quad\quad\quad\quad\quad\quad\quad \mathcal{K}(\Gamma, u{\sim}[\![A]\!]_{\mathsf{C}}; u)
\end{array}
$$

The idea is as follows. We save the current handler (at the source world w) in the variable x_{old}. When we arrive at the destination, we immediately bind a new handler for the time that we are executing there. It takes a value of type `exn` and binds it to a valid variable with `put`—this requires that the extensible type `exn` be mobile (Section 5.1.4). We then travel back to the source world and call the original handler x_{old} on the exception value. If we return from the `get` normally, we restore the handler to its original value.

First-class continuations

First-class continuations are also translated away during the conversion to CPS—in fact, this is one of the reasons we use CPS, since there is no analog to `letcc` and `throw` in one of our target languages, JavaScript.

The translation is isolated to the two constructs, and standard:

$$
\begin{array}{l}
\textsf{convert}\ \Gamma\ \mathcal{K}\ (\texttt{letcc}_A\ x\ \texttt{in}\ M)\ A\ \mathrm{w} = \\
\quad \texttt{val}\ x = \textbf{fn}(x_r).\mathcal{K}(\Gamma, x_r{:}[\![A]\!]_{\mathsf{C}}; x_r)\quad \texttt{in} \\
\quad \textsf{convert}\ (\Gamma, x{:}[\![A]\!]_{\mathsf{C}}\ \texttt{cont@w})\ \mathcal{K}'\ M\ A\ \mathrm{w} = \\
\quad \text{where}\ \mathcal{K}(\Gamma, v) = \texttt{call}\ x(v)
\end{array}
$$

For `letcc`, we reify the current continuation $\mathcal{K}$, which M might throw to. If it does not, we throw the result of M to it ourselves.

$$
\begin{array}{l}
\textsf{convert}\ \Gamma\ \mathcal{K}\ (\texttt{throw}_A\ M\ \texttt{to}\ N)\ A\ \mathrm{w} = \\
\quad \textsf{convert}\ \Gamma\ \mathcal{K}'\ M\ B\ \mathrm{w} = \\
\quad \text{where}\ \mathcal{K}'(\Gamma, v) = \textsf{convert}\ \Gamma\ \mathcal{K}''\ N\ (B\ \texttt{cont})\ \mathrm{w}
\end{array}
$$

$$\text{where } \mathcal{K}''(\Gamma, v_f) = \texttt{call } v_f(v)$$

For `throw`, we evaluate M and then N; N must be a continuation so we call it on the result of evaluating M.

Polymorphism

Polymorphism is treated simply by using the $\forall$ quantifier of the CPS language, which quantifies over worlds, types, and values. In the IL all variables are fully applied to type and world arguments, so their conversion is as follows

$$[\![x\langle A_1, \ldots, A_n; \mathrm{w}_1, \ldots, \mathrm{w}_m\rangle]\!]_{\mathsf{VC}}^{[\mathrm{w}_1,\ldots,\mathrm{w}_m/\omega_1,\ldots,\omega_m][A_1,\ldots,A_n/\alpha_1,\ldots,\alpha_n]B@\mathrm{w}} = x\langle [\![A_1]\!]_{\mathsf{C}}, \ldots, [\![A_n]\!]_{\mathsf{C}}; \mathrm{w}_1, \ldots, \mathrm{w}_m; \cdot\rangle$$

if there are no type or world arguments, then we just generate the bare variable, not an empty application. We do the same translation for valid variables.

Polymorphic variables are bound by various declarations. Conversion of IL declarations is defined by the function

$$\mathsf{convertd}\ \Gamma\ \mathcal{K}\ D\ \mathrm{w}$$

which converts D at the world w. The continuation $\mathcal{K}$ takes only the new context, since there is no value returned by a declaration.

We must do some reorganization of the polymorphic bindings so that the types work out correctly. For example, the `polyleta` construct must bind a variable at another world with polymorphic type. We translate it as

$$\begin{aligned}&\mathsf{convertd}\ \Gamma\ \mathcal{K}\ (\texttt{polyleta}\ (\alpha_1, \ldots, \alpha_n; \omega_1, \ldots, \omega_m)\ x = v)\ \mathrm{w} = \\ &\quad \texttt{leta}\ x = \mathbf{held}_{\mathrm{w}'}(\mathbf{\Lambda}\langle \omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_n; \cdot\rangle. \\ &\qquad\qquad \mathbf{leta}\ x' = [\![v]\!]_{\mathsf{VC}}^{\texttt{-at}\,\mathrm{w}'@\mathrm{w}}\ \texttt{in} \\ &\qquad\qquad x')\ \texttt{in} \\ &\quad \mathcal{K}(\Gamma; x{:}\forall\langle \omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_n; \cdot\rangle B@\mathrm{w}')\end{aligned}$$

Here is our first use of the value form for **leta**, used to expand a value of type $A\,\texttt{at}\,\mathrm{w}'@\mathrm{w}'$ to a value of type $A@\mathrm{w}'$ in place. Since the object of a `polyleta` is usually an immediate **held**, this often creates a useless eta-expansion of the value. These are reduced in an optimization phase later.

The `polyletsham` binding is similar:

$$\begin{aligned}&\mathsf{convertd}\ \Gamma\ \mathcal{K}\ (\texttt{polyletsham}\ (\alpha_1, \ldots, \alpha_n; \omega_1, \ldots, \omega_m)\ u = v)\ \mathrm{w} = \\ &\quad \texttt{letsham}\ x = \mathbf{sham}\ \omega.(\mathbf{\Lambda}\langle \omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_n; \cdot\rangle. \\ &\qquad\qquad \mathbf{letsham}\ u' = [\![v]\!]_{\mathsf{VC}}^{\mathfrak{Z}_{\omega\text{-}}@\omega}\ \texttt{in} \\ &\qquad\qquad u')\ \texttt{in} \\ &\quad \mathcal{K}(\Gamma; u{\sim}\omega.\forall\langle \omega_1, \ldots, \omega_n; \alpha_1, \ldots, \alpha_n; \cdot\rangle B)\end{aligned}$$

Network signatures

The network signature declarations (`extern`) are straightforward in the case of type and world imports. These are translated directly to their counterparts in CPS. The `extern val` declaration also has a counterpart in the CPS, but we interpret it a different way. The reason is that when we import a function, we need to use a direct-style calling convention to call it, because we cannot CPS convert external functions. Therefore, the translation of an `extern val` declaration differs based on the type of the imported value.

If A is a base type...
convertd $\Gamma\ \mathcal{K}$ (`extern val` $x : \forall\alpha_1 \ldots \alpha_n \forall\omega_1 \ldots \omega_m.A@\mathrm{w}' = \ell$) w =
  `extern val` $x : \forall\langle\alpha_1, \ldots, \alpha_n; \omega_1 \ldots, \omega_m; \cdot\rangle.[\![A]\!]_{\mathtt{c}}@\mathrm{w}' = \ell$ `in`
  $\mathcal{K}(\Gamma; x{:}\forall\langle\alpha_1, \ldots, \alpha_n; \omega_1 \ldots, \omega_m; \cdot\rangle.[\![A]\!]_{\mathtt{c}}@\mathrm{w}')$

If the value is imported at base type, then we just use the `extern val` of the CPS language, which expects base types.

If A is $\{1 : A_1, \ldots, k : A_k\} \to B$ where A_i and B are base types ...
convertd $\Gamma\ \mathcal{K}$ (`extern val` $x : \forall\alpha_1 \ldots \alpha_n \forall\omega_1 \ldots \omega_m.A@\mathrm{w}' = \ell$) w =
  `leta` x = $\mathbf{held}_{\mathrm{w}'}$
    $(\mathbf{\Lambda}\langle\alpha_1 \ldots \alpha_n; \omega_1 \ldots \omega_m; \cdot\rangle.$
    $(\mathbf{fn}(y : \{1 : [\![A_1]\!]_{\mathtt{c}}, \ldots, k : [\![A_k]\!]_{\mathtt{c}}\}, x_r : [\![B]\!]_{\mathtt{c}}\ \mathtt{cont}, \mathit{handler} : \mathtt{exn\ cont}).$
     `let` z = `primcall`$(\ell : ([\![A_1]\!]_{\mathtt{c}}, \ldots, [\![A_n]\!]_{\mathtt{c}}) \to [\![B]\!]_{\mathtt{c}})(\#1\ y, \ldots, \#k\ y)$ `in`
     $x_r\ (z)$)) `in`
  $\mathcal{K}(\Gamma; x{:}\forall\langle\alpha_1, \ldots, \alpha_n; \omega_1 \ldots, \omega_m; \cdot\rangle.(\{1 : [\![A_1]\!]_{\mathtt{c}}, \ldots, k : [\![A_k]\!]_{\mathtt{c}}\},$
    $[\![B]\!]_{\mathtt{c}}\ \mathtt{cont},$
    $\mathtt{exn\ cont})\ \mathtt{cont}@\mathrm{w}')$

If it is a function taking a record as an argument, we treat this as a function taking k arguments. We wrap a direct-style call to it using `primcall` in a continuation that uses the CPS calling convention. We do a similar thing for imported valid values and functions.

We also have a case for translating single-argument functions (so that they do not need to be imported at the unitary record type, which is awkward). Additionally, if the function returns `unit`, then we return a freshly created empty record rather than use the return value of the primcall. This is because imported JavaScript functions whose natural return type is `unit` don't actually return an empty record, they return "`undefined`."

Summary

The rest of the conversion to CPS is similar to what we have discussed or otherwise not interesting. Once in the CPS language, we immediately type-check the program to catch compiler bugs, and then engage in a series of transformations from the CPS language to itself. These transformations are all implemented using a functor for the convenient definition of type-directed transformations, which is the subject of the next section.

5.4.3 Type-directed translations

This section describes another programming idiom for implementing type directed compilers in Standard ML. It is independent from the previous one (Section 5.4.1) and not necessary for understanding the remainder of the dissertation, so it may safely be skipped.

Pointwise transformations

Very frequently, a transformation on the programming language's syntax or types is defined in a "pointwise" fashion. For example, the translation of types during closure conversion is defined as

$$\begin{aligned}
[\![(A_1, \ldots, A_n)\ \texttt{cont}]\!]_{\text{cc}} &= \exists\alpha.(\alpha, ([\![A_1]\!]_{\text{cc}}, \ldots, [\![A_n]\!]_{\text{cc}}, \alpha)\ \texttt{cont}) \\
[\![A\ \texttt{at}\ \text{w}]\!]_{\text{cc}} &= [\![A]\!]_{\text{cc}}\ \texttt{at}\ \text{w} \\
[\![\mathfrak{B}_\omega A]\!]_{\text{cc}} &= \mathfrak{B}_\omega [\![A]\!]_{\text{cc}} \\
[\![\texttt{exn}]\!]_{\text{cc}} &= \texttt{exn} \\
&\vdots
\end{aligned}$$

that is, we do a translation for the `cont` type but all of the other type constructors are handled in a uniform way, simply applying the transformation recursively and reconstructing the type. Similarly, closure conversion only needs to touch a few syntactic forms (functions, calls to functions, etc.) and works in a similar pointwise fashion for the others.

A nice way to implement these syntactic functions is to provide a single generic pointwise transformation. This allows the individual transformations to be given by only specifying the relevant cases. For example, the pointwise function on CPS types is

```
fun pointwiset (fw : world -> world) (ft : ctyp -> ctyp) typ =
  case ctyp typ of
    At (t, w) = At'(ft t, fw w)
  | Cont l = Cont' (map ft l)
  | Shamrock (wv, t) = Shamrock' (wv, ft t)
  | (* ... *)
```

It takes a function to apply to worlds, and a function to apply to types, and the type to transform. Pointwise it is not recursive; it simply applies the functions to each of the constituent parts of this level of the datatype. The implementation of a specific pointwise transformation is then just

```
fun cct typ =
  case ctyp typ of
    Cont l =>
      let val a = newvar ()
      in TExists' (a, [a, Cont' (map cct l @ [a, Cont' [a]])])
      end
  | _ => pointwiset I cct typ
```

where `newvar` creates a new variable and `I` is the identity function (on worlds). The idea is that at each level of the datatype, our conversion function is run to see if it applies; if not, it appeals back to `pointwise` to apply itself to all of the constituent parts of the constructor. Programming this way, a transformation needs only contain code for the constructs that are relevant to what it is doing. This results in shorter programs and it makes transformations work even if irrelevant constructs are added to or removed from the object language.

In ML5/pgh we have a family of `pointwise` functions for IL and CPS types, values, and expressions. They are used in a few places in ML5/pgh to briefly define purely syntactic transformations. However, because the CPS language is typed, and has type annotations within the code so that it can be effectively type-checked, the `pointwise` family turns out to be inadequate for most of the passes in the CPS phase. The `Pass` functor is a generalization to type-directed transformations that is powerful enough for us to write all of the CPS passes in the compiler.

The `Pass` functor

For a type-directed translation of expressions and values, we need more than just the raw syntax. We need to know the types of the subterms and the current typing context, and we may need additional contextual information (for example, the status of certain variables). It would be possible to arrange this as a series of functions like `pointwise`, but its type would be massively complex. Instead we use the module system, defining a functor that is used to create uniform type-directed passes over the language. With it, we will be able to specify only the cases that are relevant to our translation.

The `PASSARG` signature describes a uniform translation of the CPS language. It consists of a single function for each construct of the language:

```
signature PASSARG =
sig
  type arg

  type selves = { selfv : arg -> context -> cval -> cval * ctyp,
                  selfe : arg -> context -> cexp -> cexp,
                  selft : arg -> context -> ctyp -> ctyp }

  (* types *)
  val case_At : arg -> selves * context -> ctyp * world -> ctyp
  val case_Cont : arg -> selves * context -> ctyp list -> ctyp
  (* ... *)

  (* exps *)
  val case_Go : arg -> selves * context ->
                world * cval * cexp -> cexp
  (* ... *)
```

```
  (* vals *)
  val case_Proj : arg -> selves * context ->
                  string * cval -> cval * ctyp
  (* ... *)
end
```

The type `arg` is arbitrary, and chosen by the implementor. We can use it to pass along contextual information needed by the transformation. Each case receives an `arg`. It also receives a record containing the functions that should be used for recursive calls: one each for values, expressions, and types. Then it receives the current typing context. Finally, it has as arguments the body of the corresponding datatype constructor; the case for `At(t,w)` gets a type and a world.

Each case should return the translated construct. For types and expressions, this means returning a type or expression; for a value, we also must return its type.

From an argument module with signature `PASSARG`, the `Pass` functor produces a structure that performs the complete transformation:

```
signature PASS =
sig
  type arg
  val convertv : arg -> context -> cval -> cval * ctyp
  val converte : arg -> context -> cexp -> cexp
  val convertt : arg -> context -> ctyp -> ctyp
end
```

The implementation of the `Pass` functor is very simple; it just pattern matches on the construct and then "ties the knot" by passing the recursive function as the `selves` argument to the appropriate case:

```
  fun convertv z G va =
    let val s = { selfv = convertv,
                  selfe = converte,
                  selft = convertt }
    in
      case cval va of
        Lams a => case_Lams z (s, G) a
      | Fsel a => case_Fsel z (s, G) a
      | (* ... *)
    end
  and converte z G ex = (* ... *)
```

The trick is then as follows. Since we have exploded the case analysis into a collection of independent functions and used open recursion to recurse between them, we can now program in a style that allows us to override the behavior for a set of constructs that we choose. This is like inheritance in Object Oriented Programming languages.

We start with a `PASSARG` that implements the identity transformation. It is actually a functor that takes an `arg` type:

```
functor IDPass(type arg) : PASSARG =
struct

  fun case_Go z ({selfe, selfv, selft}, G) (w, va, e) =
  let val (va, t) = selfv z G va
  in case ctyp t of
        Addr w' => Go' (w, va, selfe z (T.setworld G w) e)
      | _ => raise Fail "go on non-addr"
  end

  (* ... *)
end
```

For each construct it does the necessary work to maintain the typing context (`T.setworld` switches the current world, so that we'll always know where we are translating). It makes recursive calls through the self arguments.

Finally, to create a pass that does something interesting, we start from the identity pass and override the cases for the constructs we are interested in. For example, here is a simple optimization pass that reduces projections from literal records.

```
structure Reduce : PASSARG =
struct
  structure ID = IDPass(type arg = unit)
  open ID

  fun case_Proj () (s as {selfv, ...}, G) (l, va) =
    case cval va of
      Record lvl =>
        (case ListUtil.Alist.find op= lvl l of
           NONE => raise Fail "bad constant proj"
         | SOME va' => selfv () G va')
    | _ => ID.case_Proj () (s, G) (l, va)

end
```

This is the entire implementation. The utility function `find` searches for the label in the record; if it is not found, the program is ill-formed. If it is, we just recurse on that value, eliminating the projection and the rest of the record. If the projection is from something other than a literal record (such as a variable), then we appeal to the function drawn from the identity pass. This appeal is similar to a call to a method on `super` in an object-oriented language.

The remainder of the functions are present in the module because we `open` the identity pass at the beginning. We then shadow the ones that we want to override. In exchange for this convenience we lose exhaustiveness checking, but this is a small price to pay for eliminating the code that would be repeated if each pass were defined over the entire language by hand.

In ML5/pgh, all of the passes of the CPS phase are implemented using the `Pass` functor or the `pointwise` functions (when they are particularly simple), except for the type checker and code generator—these both need to look at every construct, so there is no benefit to be had. Several of these passes are independent optimizations that can be run at various times during compilation; these are discussed next.

5.4.4 Optimizations

A round of optimizations is performed right after conversion to CPS, before we establish the type representation invariant (which makes optimizations more difficult to perform) or transform the program too much (which obscures high-level structure). Making the program as small as possible at this point also makes the later expensive phases faster.

The optimizations that ML5/pgh performs are not very sophisticated, but they are important for preventing the code from becoming unmanageably large. I only describe each briefly:

Dead code elimination. The most important optimization removes effectless operations whose bound variables are never used. Several other optimizations rely on this pass to clean up after them. Since the CPS language has a free variable calculation built in, this pass is easily implemented; for each eligible construct, we just check to see if the bound variable is used in the body, and eliminate it if it does not.

Inlining. It is also profitable to inline some values. If a value bound to a variable is known to be small (like an integer) or is used only once, then we can substitute it for the variable. We also non-conservatively inline some functions that appear to be good candidates, because they look (for example) like datatype constructors. Inlining is also easy to implement, because the free variable set also keeps track of how many times each variable is used. Later optimizations reduce in-line function applications.

Beta reduction. We carry out beta redices that do not increase the size of the program. For example, if we have a non-recursive literal function value applied to some values, we reduce this by binding the values to the argument variables and expanding the function body. There are often many beta redices after expanding the function this way as well. A related pass is specially tailored to run after closure conversion and remove unnecessary closures.

Eta reduction. CPS conversion can produce eta-expanded continuations, for example in the case where the last thing that a function does is call some other function. This purely syntactic transformation eta-reduces functions where possible. Without it, we would not have constant-space tail calls.

Simplification. Primitive operations like comparisons applied to constants can be simplified in the straightforward way. We also reduce `jointext` operations, which is important because string manipulation is common in our applications.

Known. Even if we decide not to inline a function or record, we can sometimes simplify operations applied to the variable it is bound to. For example, if x is bound to a record $\{\ell_1 = y, \ell_2 = v\}$ and we see a projection $\#\ell_1\ x$, we can reduce this to y. This kind of situation is particularly common for closures that escape but that have some direct calls as well. This pass uses the `arg` maintained by the `Pass` functor to collect this information about bound variables.

Unreachable code. We eliminate unreachable code (such as the default case of an exhaustive `sumcase`, which reduces code size and false dependencies on variables (especially the `Match` exception).

Here. If we have `leta` $x = \mathbf{held}_{\mathrm{w}}\ v$ and we are at the world w, then we simplify this to a regular value binding. This helps us recognize other optimization opportunities.

Many more optimizations are possible, of course, from classic ones to optimizations of our modal operations like `put` and `go`. My goal is to build a usable prototype, not a high-performance production compiler, so I have avoided spending much time on optimizations. Of the potential optimizations to add, the one that would probably have the biggest payoff would be monomorphization of needlessly polymorphic (or valid) bindings. ML5's type inference and automatic generalization are convenient, but frequently produce code that is more polymorphic than the programmer needs or wants. This is a problem only because ML5/pgh represents types at runtime in order to perform marshaling, so polymorphism and validity carry a cost. This will become evident as we begin to discuss the type representation translations in the next section.

5.4.5 Type representation

The purpose of world and type representation is to provide data that the marshaling and unmarshaling procedures can use to manipulate data of polymorphic type. Providing this information is a two-stage process that brackets closure conversion: We begin by establishing an invariant with a pass over the code, closure conversion requires and maintains this invariant, and a final pass uses the invariant to create representations. After this pass we still have types and worlds, but they are purely static entities.

The invariant that we establish is as follows: For any type variable α in scope, we also have a valid variable of type α `rep` in scope. The variable must be valid because a type is not tied to any particular world, and so we might use that type at any world where it is in scope. We also have a corresponding invariant for worlds: for any world variable ω in scope, we also have a valid variable of type ω `wrep` in scope.

This first phase simply establishes this invariant. To do so, we consider each of the terms and values that bind world or type variables. We are not concerned with

type-level constructs that bind variables, such as μ, because these type variables cannot appear in terms. The relevant constructs are therefore $\mathbf{\Lambda}$, which binds type and world variables; **sham**, which binds a world variable; and `extern type`, which binds a type variable. The `unpack` and **unpack** constructs for existential types also bind a type variable, but we do not expect to see these at this phase of the compiler, because they are introduced during closure conversion.

The way that we establish the invariant is to augment each of these three so that they bind the representation variables in addition to the type and world variables. Each is handled in a somewhat different manner. The $\mathbf{\Lambda}$ construct abstracts over worlds, types, and values. This gives us a convenient place to put the representations;

$$\mathbf{\Lambda}\langle \alpha_1, \ldots, \alpha_n;\ \omega_1, \ldots, \omega_m;\ x_1{:}A_1, \ldots, x_k{:}A_k\rangle.v$$

is translated to

$$\begin{array}{l}
\mathbf{\Lambda}\langle \alpha_1, \ldots, \alpha_n;\ \omega_1, \ldots, \omega_m; \\
\quad x_1{:}A_1, \ldots, x_k{:}A_k, \\
\quad y_1{:}\mathfrak{B}(\alpha_1\ \texttt{rep}), \ldots, y_n{:}\mathfrak{B}(\alpha_n\ \texttt{rep}), \\
\quad z_1{:}\mathfrak{B}(\omega_1\ \texttt{wrep}), \ldots, z_m{:}\mathfrak{B}(\omega_m\ \texttt{wrep})\rangle. \\
\quad \mathbf{letsham}\ u_{\alpha_1} = y_1\ \texttt{in} \\
\qquad\qquad\qquad \vdots \\
\quad \mathbf{letsham}\ u_{\alpha_n} = y_n\ \texttt{in} \\
\quad \mathbf{letsham}\ u_{\omega_1} = z_1\ \texttt{in} \\
\qquad\qquad\qquad \vdots \\
\quad \mathbf{letsham}\ u_{\omega_m} = z_m\ \texttt{in}\ v
\end{array}$$

by adding one value argument for each of the type and world arguments. The values are wrapped in the $\mathfrak{B}$ modality so that they can be bound as valid variables in the body. We use the value form of **letsham** because the body is a value. From this transformation the type

$$\forall\langle \alpha_1, \ldots, \alpha_n;\ \omega_1, \ldots, \omega_m;\ A_1, \ldots, A_k\rangle.B$$

becomes

$$\begin{array}{l}
\forall\langle \alpha_1, \ldots, \alpha_n;\ \omega_1, \ldots, \omega_m;\ A_1, \ldots, A_k, \\
\quad \mathfrak{B}(\alpha_1\ \texttt{rep}), \ldots, \mathfrak{B}(\alpha_n\ \texttt{rep}), \\
\quad \mathfrak{B}(\omega_1\ \texttt{wrep}), \ldots, \mathfrak{B}(\omega_m\ \texttt{wrep})\rangle.B
\end{array}$$

We must also transform applications of $\forall$ type, since they would otherwise not pass enough arguments. We do so by using the stand-in **repfor** and **wrepfor** values. So

$$v\ \langle A_1, \ldots, A_n;\ \mathrm{w}_1, \ldots, \mathrm{w}_m;\ v_1, \ldots, v_k\rangle$$

is transformed to

$$\begin{array}{l}
v\ \langle A_1, \ldots, A_n; \mathrm{w}_1, \ldots, \mathrm{w}_m; v_1, \ldots, v_k, \\
\quad \mathbf{sham}\ (\mathbf{repfor}\ A_1), \ldots, \mathbf{sham}\ (\mathbf{repfor}\ A_n), \\
\quad \mathbf{sham}\ (\mathbf{wrepfor}\ \mathrm{w}_1), \ldots \mathbf{sham}\ (\mathbf{wrepfor}\ \mathrm{w}_n)\rangle
\end{array}$$

This both establishes and uses the invariant: If in **repfor** A the type A contains a type variable α, our ability to generate an actual representation at this point will rely on there being a representation of α in scope.

The `extern type` import is very simple; since we can know nothing about this type, we just require that the import be accompanied by a valid value that is its representation. We expect by convention that this value is associated with a label derived from the type's label. Therefore

$$\texttt{extern type}\ \alpha = \ell\ \texttt{in}\ c$$

is translated to

$$\texttt{extern type}\ \alpha = \ell\ \texttt{with rep}\ u_\alpha = \ell_{\texttt{rep}}\ \texttt{in}\ c$$

where $\ell_{\texttt{rep}}$ is the arbitrary but agreed upon way of deriving the type representation's label from the type's label.

Finally, the value of shamrock type is a little trickier. We transform

$$\mathbf{sham}\ \omega.v$$

to

$$\mathbf{sham}\ \omega.\mathbf{\Lambda}\langle\cdot;\cdot;x{:}\mathfrak{B}(\omega\ \texttt{wrep})\rangle.\ \mathbf{letsham}\ u_\omega = x\ \texttt{in}\ v$$

and so the type

$$\mathfrak{B}_\omega A$$

becomes

$$\mathfrak{B}_\omega\forall\langle\cdot;\cdot;\mathfrak{B}(\omega\ \texttt{wrep})\rangle.A$$

This is straightforward except for the elimination form. Because `letsham` binds a variable, we must rewrite not `letsham` itself but uses of `letsham`-bound valid variables. (We also do a similar translation for the value form, **letsham**.) Therefore,

$$\texttt{letsham}\ u = v\ \texttt{in}\ c$$

becomes

$$\texttt{letsham}\ u = v\ \texttt{in}\ (\mathsf{inst}\ u\ \mathrm{w}\ c)$$

where the function inst instantiates uses of u within c. For continuation expressions, inst just finds values and applies instv to them, keeping track of the current world. The function instv $u\ \mathrm{w}\ v$ is pointwise in v (also tracking the current world) except on a matching valid variable:

$$\mathsf{instv}\ u\ \mathrm{w}\ u = u\ \langle\cdot;\cdot;\mathbf{sham}\ (\mathbf{wrepfor}\ \mathrm{w})\rangle$$

This part of the translation is implemented by maintaining a set of valid variables that were bound by `letsham` or **letsham**. This set is the `arg` that is threaded through the translation by the `Pass` functor.

This suffices to establish the type and world representation invariant. We must now be somewhat careful when working on the code; for example, the dead code optimization is unsafe because it might throw away apparently unused type representations. This will be particularly delicate during closure conversion, which is the next phase.

5.4.6 Closure conversion

In the closure conversion phase we will establish another invariant, which is that every **fns** and $\mathbf{\Lambda}$ in the program is closed to modal and valid variables. We use this invariant in the hoisting phase (Section 5.4.8) to pull each piece of code out to the top level so that we can refer to them by global labels. In this phase we also translate `go` to take a closure as an argument rather than a literal continuation, so that we can later marshal this closure as data to be sent over the network.

This translation is complicated for a number of reasons. To highlight the important aspects, I will start by giving the translation for the $\mathbf{\Lambda}$ construct, since it is non-recursive and exposes all of the issues particular to the modal setting. I then explain how it is extended to the mutually-recursive **fns** value. I will then discuss the direct call idiom, which is an optimization to produce substantially better code in the case that functions do not escape.

The heart of closure conversion is building an environment that contains the free (dynamic) variables of a function, and then passing that environment as an additional argument to the function. The body of the function then receives these free variables from the explicit environment, rather than its surrounding context, making it closed. The first complication in ML5/pgh is that dynamic variables include modal variables bound at other worlds, and valid variables. We already discussed the solution to this in the previous chapter; we use the `at` and $\mathfrak{S}$ modalities to encapsulate these variables in the environment. Therefore the environment for an abstraction $\mathbf{\Lambda}\langle\vec{\alpha};\vec{\omega};\overline{x:A}\rangle.v$ will take the form

$$\mathbf{held}_{\mathrm{w}_1}\ x_1, \ldots, \mathbf{held}_{\mathrm{w}_n}\ x_n;\ \mathbf{sham}\ \omega.u_1, \ldots, \mathbf{sham}\ \omega.u_m$$

where x_i are the free modal variables of type $A_i@\mathrm{w}_i$ and u_j are the free valid variables of type $\omega.A_j$.

The next consideration is the maintenance of our type representation invariant. In order to preserve the invariant that each type and world in scope also has a valid representation variable, we must also include those valid representation variables in our environment. Naïvely, this would include every type variable in scope. This would make environments much larger than they needed to be. Therefore, we relax this to those representation variables that we might actually need. Which ones might we need? This clearly includes the representation of any type or world variable that literally occurs free in the body of the function. It is not enough, however, to just consider these—for example, the $\mathbf{\Lambda}$ might contain within it the continuation

$$\texttt{go}[\mathrm{w}', v]\ \texttt{call}\ v_f\ (v')$$

where v' is free. This has no literally occurring type variables. However, compilation of this will require us to marshal v_f and v'. If one of these has a type involving a free type variable α, then we will need its representation. Therefore, the right way to think of the free type variables is as the variables of the *typing derivation* rather than the CPS term. It suffices to take the set of actually occurring type and world variables, unioned with the type and world variables that occur in the types and worlds of the free modal and valid variables. We potentially need the representation for each of these types and worlds.

There is one more subtlety to the computation of the free variable set. Just as we should not consider a variable free if it is an argument to a function, a representation variable need not be considered free if there is already a representation for that type being passed to the function that we are closure converting. For example, if we have

$$\mathbf{\Lambda}\langle\cdot;\cdot;x{:}\mathbb{B}(\omega\ \texttt{wrep})\rangle.\ \mathbf{letsham}\ u_\omega = x\ \texttt{in}\quad \ldots\ \omega\ \ldots$$

it may appear that we require a representation for ω, because it appears free in the body of the $\mathbf{\Lambda}$. However, we already have a representation variable in scope for the body! It is wasteful to pass another copy, but not doing so is also required for correctness in some cases. This is because we use the same constructs that we are closure-converting in order to establish the invariant in the first place. In particular, in the previous phase the $\mathbf{sham}\ \omega.v$ construct was translated to

$$\mathbf{sham}\ \omega.\mathbf{\Lambda}\langle\cdot;\cdot;x{:}\mathbb{B}(\omega\ \texttt{wrep})\rangle.\ \mathbf{letsham}\ u_\omega = x\ \texttt{in}\ v$$

It would be a mistake to require a representation for ω in the closure for the $\mathbf{\Lambda}$ here. If we did so, then we would need to build an environment between the $\mathbf{sham}$ and the $\mathbf{\Lambda}$ that contained a representation for ω. However, we have no such representation to use, because it is the argument to the $\mathbf{\Lambda}$ that is intended to provide the representation of the world variable ω that was just bound.

Therefore, the translation for $\mathbf{\Lambda}\langle\vec{\alpha};\vec{\omega};\overline{y{:}A}\rangle.v$ is as follows. Let $y_i{:}A_i@\mathrm{w}_i$ be the free modal variables of the $\mathbf{\Lambda}$, and $u_j{\sim}\omega.{:}B_j$ be the free valid variables. We then let the representable type variables α_k be the be the free type variables of the typing derivation for v, minus any among the arguments $\vec{\alpha}$, and minus any that appear as the type $\mathbb{B}(\alpha\ \texttt{rep})$ of some value argument y. The representable world variables ω_l are defined the same way. By our invariant, for each representable type and world variable we have a valid variable u_α or u_ω that is its representation in scope. These are added to the set of free valid variables u_j above. Finally, the environment env is

$$\begin{aligned}\{&\ell_{y_1} = \mathbf{held}_{\mathrm{w}_1}\ y_1,\ \ldots,\\ &\ell_{y_n} = \mathbf{held}_{\mathrm{w}_n}\ y_n,\\ &\ell_{u_1} = \mathbf{sham}\ \omega.u_1,\ \ldots,\\ &\ell_{u_m} = \mathbf{sham}\ \omega.u_m\}\end{aligned}$$

and the environment type $envt$ is

$$\begin{aligned}\{&\ell_{y_1} : A_1\ \texttt{at}\ \mathrm{w}_1,\ \ldots,\\ &\ell_{y_n} : A_n\ \texttt{at}\ \mathrm{w}_n,\\ &\ell_{u_1} : \mathbb{B}_\omega B_1,\ \ldots,\\ &\ell_{u_m} : \mathbb{B}_\omega B_m\}\end{aligned}$$

The newly-closed $\mathbf{\Lambda}$, which we'll call v_l, projects out the components:

$$\begin{array}{l}\mathbf{\Lambda}\langle\vec{\alpha};\vec{\omega};\overline{y{:}A},x_e{:}envt\rangle.\\ \quad\mathbf{leta}\ y_1 = \#\ell_{y_1}\ x_e\ \texttt{in}\\ \qquad\qquad\vdots\\ \quad\mathbf{leta}\ y_n = \#\ell_{y_n}\ x_e\ \texttt{in}\\ \quad\mathbf{letsham}\ u_1 = \#\ell_{u_1}\ x_e\ \texttt{in}\\ \qquad\qquad\vdots\\ \quad\mathbf{letsham}\ u_m = \#\ell_{u_m}\ x_e\ \texttt{in}\\ \quad v\end{array}$$

And finally, the closure is the environment and the function, packed into an existential so that the type of closures is uniform:

$$\begin{array}{l}\mathbf{pack}\ envt;\ \mathbf{sham}\ (\mathbf{repfor}\ envt);\ env, v_l\\ \mathbf{as}\ \exists\alpha.(\alpha, \forall\langle\vec{\alpha};\vec{\omega};\vec{A},\alpha\rangle.B\end{array}$$

which has the type that the annotation indicates. Recall that a **pack** always takes a representation for the hidden type variable, which is embedded in the $\Box$ modality to ensure its validity.

Of course, the types that are part of the annotations undergo a translation as well, and the closure conversion transformation is applied recursively so that the any functions within the body v are also closure converted; we omit that here to reduce distractions.

In addition to translating the functions into closures, we must also translate the call sites so that they use the closure to make the call. An application

$$v_f\ \langle\vec{A};\vec{\mathrm{w}};\vec{v}\rangle$$

is translated to the following value

$$\begin{array}{l}\mathbf{unpack}\ \alpha;\ u_\alpha;\ x_e{:}\alpha, x_f{:}\forall\langle\vec{\alpha};\vec{\omega};\vec{B}\rangle.C = v_f\ \texttt{in}\\ x_f\ \langle\vec{A};\vec{\mathrm{w}};\vec{v}, x_e\rangle\end{array}$$

where, after unpacking, we are just passing along the environment as an additional final argument.

Mutual recursion

The mutually recursive **fns** construct is closure-converted in a similar way. To simplify the discussion, let us pretend that each function in the bundle takes a single argument:

$$\mathbf{fns}(f_1(x_1) = c_1;\ \ldots;\ f_n(x_n) = c_n)$$

We compute a single environment for the entire bundle of functions. The free variables y_i and u_j, as well as the environment *env* and the environment type *envt* are computed the same way as before.[5] Inside each c_k, we begin by binding all of the free variables from the environment in the familiar way. This leaves us with only thing left, which is to account for the mutual recursion between the functions.

Within c_i, we may use any of the functions f_i ("friend" functions)—we could call them (as closure calls) or pass them off to other functions. Therefore, after restoring the free variables in the continuation body we then make a closure for each of the friends. These are optimized away in the case that they are not used.

$$\begin{array}{l}
\texttt{val}\, f_1 = \mathbf{pack} envt;\ \mathbf{sham}\ (\mathbf{repfor}\ envt);\ x_e, f_1\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\texttt{val}\, f_n = \mathbf{pack} envt;\ \mathbf{sham}\ (\mathbf{repfor}\ envt);\ x_e, f_n\ \texttt{in} \\
c_i
\end{array}$$

We use the recursive variable bound by the **fns** construct, but wrap it as a closure, using the same environment x_e that was an argument to this function in the bundle.

The whole **fns** itself is packed into an existential, along with the environment, as before. Because we select a function out of a bundle with **fsel** before calling it, we also have to translate that. The value

$$\mathbf{fsel}\ v.n$$

becomes the value

$$\begin{array}{l}
\mathbf{unpack}\ \alpha;\ u_\alpha;\ x_e{:}\alpha, x_f{:}((A_1, \alpha), \ldots (A_m, \alpha))\ \texttt{conts} = v\ \texttt{in} \\
\mathbf{pack}\ \alpha;\ u_\alpha; x_e, \mathbf{fsel}\ x_f.n \\
\mathbf{as}\ \exists\beta.(\beta, (A_n, \beta)\ \texttt{cont})
\end{array}$$

It is frequently the case that an **fsel** is applied to a literal **fns** value. (This is particularly common when there is just a single function in the bundle.) In this case we will produce a spurious pack/unpack/pack sequence; one of the optimization phases reduces this.

A call to a continuation unpacks the closure and calls it on its argument and environment in the obvious way; I do not give the translation here.

Go

Aside from the **Λ** and **fns** constructs, we also closure convert the body of a `go`, so that we can treat it as a piece of data to later be marshaled. This is accomplished simply by pretending that it is actually a **fns** bundle of a single function taking zero arguments. We translate the `go` to a `go_cc`, which expects a closure.

[5] We don't exempt a type or world on account of its representation already being among the arguments. For one thing, it would have to be an argument to all of the functions in order for this to be safe. More importantly, representations never appear as arguments to **fns** at this stage of the compiler anyway.

Direct calls

The closure conversion algorithm that I have described here is correct and general, but very simplistic. In fact, most functions in most programs are used in a stylized way, where the function is defined and then called several times within its scope, but never placed inside a data structure or passed as an argument to another function. For functions used in this idiomatic way, building an environment is more costly than is necessary. (One of the big costs is that we need to store type representations even when we are not using polymorphism in the source program, because of the existential type.)

Another form of closure conversion, which we can apply selectively instead of the above strategy, is the *direct call* calling convention. After computing the free variable set, we just add these variables as additional parameters to the function rather than putting them in an explicit environment. Because the function does not escape, each of the calls to it also has those free variables in scope, so we just augment the call with the variables as additional arguments. This works for recursive and mutually-recursive functions as well, as long as the function does not escape within its own (or a friend's) body.

I implement the direct call calling convention for a selection of common cases, but it ought to be extended to more. The way that we bind functions using a variety of different binders (`val`, `letsham`, `leta`) and the fact that a function may be polymorphic makes it difficult to recognize all instances of it. (The implementation is burdened by having to maintain the type representation invariant, as well.) However, it is worthwhile to try, because when it applies it substantially reduces code size for that function and results in noticeably better performance.

Having completed closure conversion, we now know all of the places where type representations might be used. We therefore want to proceed immediately to producing real representations for them. This allows us to discard the invariant and begin treating data as data and types as purely static.

5.4.7 Type representation II

Type and world representations will be used wherever we perform a `go_cc`, to marshal the closure so that it can be sent to the remote world. We now have a complete picture of all of the places where type and world representations are generated (as the occurrences of **repfor** and **wrepfor**). We also have ensured that at any `go_cc` we can get the representation of any type or world we need. The job of this next phase is to insert the actual representations, and to insert the uses of `marshal` so that we can use the low-level `go_mar` construct. These two are essentially separable; I will discuss the former first.

Generating representations. For each **wrepfor** in the program, we perform a simple translation. If if is **wrepfor w**, then we replace this with **wrep w**. This is the only closed value of `wrep` type. If it is **wrepfor** ω, then we translate this to u_ω, where u_ω is the valid

representation variable associated with ω by invariant. Since worlds have such simple structure, this is all that is needed.

For types it is only a little more complicated. At each **repfor**A value, we recurse over the structure of A. At a type variable α we return the associated u_α. Worlds can appear in types as well, so at a world w we appeal to the above as if we saw **wrepfor** w. For any type constructor, we recurse to produce the representations of the components, and then use the corresponding representation constructor. For example, to compute the representation of

$$\{\ell_1 : A_1, \ell_2 : A_2 \,\mathtt{at}\, \mathrm{w}\}$$

we compute v_1, v_2, and v_w, which are the representations of A_1, A_2, and w, and then produce

$$\mathbf{rep}(\{\ell_1 : v_1, \ell_2 : \mathbf{rep}(v_2 \,\mathtt{at}\, v_w)\}$$

The only last complication is types that bind type or world variables. For example, when translating

$$\forall\langle\alpha; \cdot; \cdot\rangle.A$$

we will produce a representation of the form

$$\mathbf{rep}(\forall\langle(\alpha, u_\alpha); \cdot; \cdot\rangle.v)$$

where v is the representation of A. A may mention α, however, and should use the representation u_α in that case; we therefore just bind $u_\alpha{\sim}\alpha$ `rep` in the context as we recurse and it works as desired.

Using marshal. The `go_cc` construct takes an address, (remote) environment, and (remote) closure-converted function as an argument:

$$\mathtt{go\_cc}[\mathrm{w}', v_a, v_e, v_f]$$

To translate this, we will use `marshal` to produce a marshaled value of type `bytes` that we can then use with `go_mar`. At the other side—code that we will not write until we look at the runtime system in Section 5.5.4—we will receive the bytes, unmarshal them, and invoke the code on the environment. At the remote world, we will need to know the type of the code and environment, so the natural thing to do is to wrap them in an existential package so that the type is always the same. We can then pass a constant representation to the unmarshal procedure. (This works because values of existential type *contain* the dynamic representation of the hidden type, so we end up sending the representation that way.)

Suppose the type of the environment v_e is *envt*. Then, the code we generate is

$\mathtt{val}\ p = \mathbf{held}_{\mathrm{w}'}(\mathbf{pack}\ envt; \mathbf{repfor}\, envt; v_e, v_f$
$\qquad\qquad \mathbf{as}\ \exists\alpha.(\alpha, \alpha\mathtt{cont}))\ \mathtt{in}$
$\mathtt{val}\ b = \mathtt{marshal}\ (p, \mathbf{repfor}\ ((\exists\alpha.(\alpha, \alpha\mathtt{cont}))\,\mathtt{at}\,\mathrm{w}'))\ \mathtt{in}$
$\mathtt{go\_mar}[\mathrm{w}', v_a, b]$

where the **repfor** is expanded to a real representation as above. Therefore, at any given world w we always unmarshal at the type $(\exists\alpha.(\alpha, \alpha\mathtt{cont}))\,\mathtt{at}\,\mathrm{w}$.

Representing representations. A natural question is, now that we have generated these type representations, what form will they actually take at runtime? In fact, the abstract representation we are using here is totally general, in that the representation is isomorphic to the type itself. The actual representations that we will use (discussed in Section 5.4.9) will not contain as much information—for instance, we will marshal every arrow type the same way, so we only need to store enough to know that it is an arrow type. By using this most general representation, we allow ourselves the freedom to choose any "quotient" of it at the point we do code generation. (As a disadvantage of leaving this decision to the last minute, we may end up keeping around more dynamic information than we will ultimately need to use, however.)

Each world has a single representation, just as each world has a single address. In fact, these will be the same at runtime. One consequence of our world representation phase is that we could have provided the programmer not only with a `localhost` operation for computing one's own address, but an `addressof` operation that produced the dynamic address for a static world variable. This is probably not important for the current prototype, where there are only two worlds that we already know the identity of, but might be a useful feature for a future iteration.

After inserting type and world representations, we perform dead code optimizations to remove them, if possible. We also attempt to optimize poor closure conversions now, since these optimizations are easier to do when they do not need to be sensitive to the type representation invariant. Once the code has been cleaned up, it can finally be hoisted, which is the last major phase before code generation.

5.4.8 Hoisting

The purpose of hoisting is to assign each closed piece of code a unique global label, so that we can use these labels to refer to the code. The labels will later be compiled as integers, so that calling a function will consist of looking up the code in an array and jumping to it. We will also use these labels as the marshaled format of code on the network; all of the worlds will agree upon the set and meaning of these integers.

Since all of the code will be arranged in an array at the top level, we need to pull any nested code out of its scope. Because of closure conversion, each piece of code is closed to any dynamic (modal and valid) variables. However, it may still have free static (type and world) variables. As we pull code to the top level, we will need to abstract over these static variables, so that the code remains well-formed. At this stage, we also collect all of the world constants that are declared in the program with `extern val`. (This is not really related to hoisting but it is a convenient place to do it.)

To represent the hoisted program, we need new constructs in the CPS language. A program P is a set of labels (each associated with a *global* G, which is some code with its type), a set of worlds (each associated with its worldkind) and a distinguished label

from the set, which is the entry point of the program:

$$\begin{array}{ll}\{ & \textbf{labs}\ \ell_1 = G_1, \ldots, \ell_n = G_n \\ & \textbf{worlds}\ \mathbf{w}_1/J_m, \ldots, \mathbf{w}_m/J_m \\ & \textbf{main}\ = \ell \qquad\qquad\qquad\qquad \}\end{array}$$

A global is either a modal value or a valid value:

$$\begin{array}{rcl} G & ::= & v_i : A_i@\mathbf{w}_i \\ & | & \omega.v_j \sim \omega.A_j \end{array}$$

In either case, the type of the global will either be $\forall\langle\vec{\alpha};\vec{\omega};\cdot\rangle.((\vec{A_1}),\ldots,(\vec{A_n}))$ `conts` or $\forall\langle\vec{\alpha};\vec{\omega};\cdot\rangle.\forall\langle\vec{\alpha'};\ \vec{\omega'};\vec{A}\rangle.B$, since we hoist only the **fns** and $\mathbf{\Lambda}$ constructs. They each will be abstracted over some type and world variables so that they can be hoisted from their static contexts. To type check the program, we introduce two new kinds of hypotheses that can appear in the context:

$$\begin{array}{rcl} \ell_i & : & A_i@\mathbf{w}_i \\ \ell_j & \sim & \omega.A_j \end{array}$$

for modal and valid globals, respectively. Since the globals may refer to one another recursively, we begin by producing the context for type-checking the program, by assuming that the type annotations are correct. It is defined straightforwardly on the list L of label/global bindings.

$$\begin{array}{rcl} \mathsf{ctxfor}(\cdot) & = & \cdot \\ \mathsf{ctxfor}(\ell = v : A@\mathbf{w}, L) & = & \mathsf{ctxfor}(L),\ \ell : A@\mathbf{w} \\ \mathsf{ctxfor}(\ell = \omega.v \sim \omega.A, L) & = & \mathsf{ctxfor}(L),\ \ell \sim \omega.A \end{array}$$

Then, checking a program is simple:

$$\frac{\begin{array}{ll}\mathsf{ctxfor}(L) = \Gamma & \\ \Gamma, \omega\ \mathsf{world} \vdash v_i : A_i@\omega & (\text{when } L_i\ =\ \ell_i = \omega.v_i \sim \omega.A_i) \\ \Gamma \vdash v_i : A_i@\mathrm{w}_i & (\text{when } L_i\ =\ \ell_i = v_i : A_i@\mathrm{w}_i) \\ \Gamma \vdash v_i : \forall\langle\cdot;\cdot;\cdot\rangle.(())\ \texttt{conts@home} & (\text{for the } i \text{ where } L_i\ =\ \ell = v_i : A_i@\mathrm{w}_i)\end{array}}{\vdash \{\textbf{labs}\ L;\ \textbf{worlds}\ \mathbf{w}_1/J_m, \ldots, \mathbf{w}_m/J_m;\ \textbf{main}\ = \ell\}\ \mathsf{ok}}$$

We check that every label has the type its annotation indicates, assuming that the others are correct. We also check that the main label is a bundle with a single continuation that takes no arguments. The code makes reference to other code by using labels as values. There are two typing rules for labels, since they might be modal or valid:

$$\frac{}{\Gamma, \ell : A@\mathrm{w}, \Gamma' \vdash \ell : A@\mathrm{w}}$$

$$\frac{}{\Gamma, \ell \sim \omega.A, \Gamma' \vdash \ell : [{}^{\mathrm{w}}/_{\omega}]A@\mathrm{w}}$$

Translation

With this setup, the translation is easy. We crawl over the input expression (using the `Pass` functor) while maintaining a partially-built program that we update imperatively. When we see an `extern world` declaration, we insert that world with its worldkind into the set of worlds, if it is not already there. When we see a **fns** or **Λ**, we first translate its subterms recursively. We then compute the free type variables $\vec{\alpha}$ and world variables $\vec{\omega}$ of its typing derivation. This will be a modal global if its world is a constant; otherwise, it is a world variable ω_0 that must be part of the set $\vec{\omega}$. If modal, we abstract over all of the free world and type variables to produce the following global:

$$\ell = \mathbf{\Lambda}\langle\vec{\alpha};\vec{\omega};\cdot\rangle.v : \forall\langle\vec{\alpha};\vec{\omega};\cdot\rangle.B$$

where v is the original value and B is its type. ℓ is a freshly-generated label. Since we have abstracted over all of the value's free static variables and it has been closure converted, this global is well-formed in the empty context.

Having inserted the global into the program, we then replace its occurrence in the expression with ℓ, applied to all of the static variables that we abstracted over:

$$\ell\ \langle\vec{\alpha};\vec{\omega};\cdot\rangle$$

This gives it the same type that it originally had.

For valid globals, it is only slightly trickier. We do not abstract over the world variable ω_0 (the world of the value) because the type-level $\forall$ quantifier does not extend to the judgment. Call $\vec{\omega'}$ the set of free static variables other than ω_0 (*i.e.* $\vec{\omega'} = \vec{\omega} - \omega_0$). We produce the global

$$\ell = \omega_0.\mathbf{\Lambda}\langle\vec{\alpha};\vec{\omega'};\cdot\rangle.v \sim \omega_0.\forall\langle\vec{\alpha};\vec{\omega'};\cdot\rangle.B$$

The use of the global is replaced with

$$\ell\ \langle\vec{\alpha};\vec{\omega'};\cdot\rangle$$

where the world ω_0 is filled in by the typing rule for valid labels.

Main. The input to the hoisting phase is a CPS expression. To produce a program, which is just a list of labeled globals, we also pretend as though the whole expression is itself a zero-argument continuation value to be hoisted. It will already be closed in the empty context, but we abstract over the empty list of world and type variables for uniformity. After it has been recursively translated, we insert the whole CPS expression into the program at a freshly generated label, and then record that as the **main** label.

Optimization. When we insert code into the program, we need not necessarily generate a fresh label. If equivalent code of the same type already exists in the program, we can re-use that label instead, coalescing the two. Because the CPS implementation gives us alpha-conversion respecting comparison between values, this is easy to implement. However, since labels are not variables, we might not recognize some collections

of mutually-recursive code that are equivalent up to the choice of labels without using a more sophisticated algorithm.

After hoisting is complete, we can optimize and type check the program for a final time, and then proceed to code generation.

5.4.9 Code generation

We are now ready to generate code for the languages that our compiler will output to. (We do not need to do any traditional compiler back-end tasks like register allocation because our target languages are currently very high-level.) Each platform has its own code generator, though they are similar. A simple driver takes the hoisted program and invokes the code generators for each of the worlds involved in the program.

All of the hosts need to agree on the program that they will run and a marshaled format for the data that they share. On each host the code globals are arranged in an array, where the indices are the same across hosts—this allows for hosts to marshal a piece of code as the same global index for everyone. However, not every code global will be generated for every host—for the world $\mathbf{w}$, a modal code global typed at a different world $\mathbf{w}'$ will not be generated. (In fact, there may be no way to sensibly generate the code, since it may make reference to local resources at $\mathbf{w}'$.) This means that the array of code on any given host contains gaps wherever the code global is specific to some different world. Valid globals are always generated for every host.

For a program $\{\mathbf{labs}\ L,\ \mathbf{worlds}\ W,\ \mathbf{main} = \ell\}$, the driver arbitrarily orders L and assigns consecutive integer indices to the labels. Then, for each world constant $\mathbf{w}$ with worldkind J in W, it invokes the code generator for J on each of the modal labels at $\mathbf{w}$ and all of the valid labels to populate the array. Each code generator also has a way of marking the gaps in the array for labels at other worlds.

Currently there are two code generators, the Bytecode language for the server, and JavaScript for the web client. Since the Bytecode language was designed for this task, the code generator is much simpler. I will discuss it first.

Bytecode code generator

The bytecode B5 is a simple untyped interpreted language based on the ML5 CPS language. Its syntax appears in Figure 5.17; there are statements (which roughly correspond to CPS expressions), expressions and values (which correspond to CPS values) and globals. A global can be of one of three forms: absent, meaning that there is no code for this host at this index; a function global, which is a list of statements each with a list of parameters (*i.e.*, a bundle of functions); or a value global, which is a statement taking a list of parameters. Since this language has no static types, these parameters are all value variables, and there is no distinction between modal and valid variables. A function global ends with a `call` to another function or a by halting. A value global (implementing a hoisted $\mathbf{\Lambda}$) ends with a `return` of some value. Such values are guaranteed to be pure and always return; we instantiate them with the expression $e\ \langle e_1, \ldots, e_n \rangle$.

$$
\begin{array}{rrcl}
\text{vals} & v & ::= & i \mid \texttt{"string"} \mid \mathbf{inj}_\ell\, v \mid \mathbf{inj}_\ell\, — \\
& & \mid & \{\!\{\ell_1 = v_1,\ \ldots,\ \ell_n = v_n\}\!\} \mid \mathbf{ptr}\ \rho \mid \mathbf{tag}\ \texttt{"addr"}, i \\
& & \mid & \mathbf{rep}(\{\ell_1 : v_1,\ \ldots,\ \ell_n : v_n\}) \\
& & \mid & \mathbf{rep}(\exists\gamma.v_1,\ \ldots,\ v_n) \\
& & \mid & \mathbf{rep}([\ell_1 : v_1^?,\ \ldots,\ \ell_n : v_n^?]) \\
& & \mid & \mathbf{rep}(v_1\ \texttt{at}\ v_2) \\
& & \mid & \mathbf{rep}(\forall\langle\gamma_1,\ \ldots,\ \gamma_n; \gamma'_1,\ \ldots,\ \gamma'_m; \cdot\rangle.v) \\
& & \mid & \mathbf{rep}(\mathfrak{B}\gamma.v) \\
& & \mid & \mathbf{rep}(\pi_i(\mu(\gamma_1.v_1,\ \ldots,\ \gamma_n.v_n))) \\
& & \mid & \mathbf{rep}\ \gamma \mid \mathbf{rep}\ r \mid \mathbf{wrep}\ \texttt{"addr"} \\
\\
\text{exps} & e & ::= & v \mid x \mid \texttt{inj}_\ell\, e \mid e\ \langle e_1,\ \ldots,\ e_n\rangle \\
& & \mid & \texttt{primcall}\ \ell(e_1, \ldots, e_n) \\
& & \mid & \texttt{native}\ p(e_1, \ldots, e_n) \\
& & \mid & \texttt{newtag} \mid \texttt{marshal}\ e, e_r \\
& & \mid & \#\ell\ e \mid \{\ell_1 = e_1,\ \ldots,\ \ell_n = e_n\} \\
& & \mid & \texttt{rep}(\{\ell_1 : e_1,\ \ldots,\ \ell_n : e_n\}) \\
& & \mid & \texttt{rep}(\exists\gamma.e_1,\ \ldots,\ e_n) \\
& & \mid & \ldots \\
\\
\text{optional vals} & v^? & ::= & v \mid — \\
\\
\text{rep vars} & \gamma & & \\
\\
\text{prim reps} & r & ::= & \mathbf{Rcont} \mid \mathbf{Rconts} \mid \mathbf{Raddr} \mid \mathbf{Rrep} \mid \mathbf{Rwrep} \mid \mathbf{Rint} \\
& & \mid & \mathbf{Rstring} \mid \mathbf{Rvoid} \mid \mathbf{Rall} \mid \mathbf{Rref} \mid \mathbf{Rexn} \mid \mathbf{Rtag} \\
\\
\text{statement} & s & ::= & \texttt{val}\ x = e\ \texttt{in}s \\
& & \mid & \texttt{call}\ e_1.e_2(e'_1, \ldots, e'_n) \\
& & \mid & \texttt{halt} \mid \texttt{return}\ e \\
& & \mid & \texttt{go}[e_a, e_b] \\
& & \mid & \texttt{case}\ e\ \texttt{of}\ x.(\ell_1 \Rightarrow s_1 \mid \ldots \mid ell_n \Rightarrow s_n \mid \_ \Rightarrow s) \\
& & \mid & \texttt{untag}\ e_o\ \texttt{with}\ e_t\ \texttt{of}\ (\texttt{yes}\ \Rightarrow x.s \mid \texttt{no} \Rightarrow s') \\
\\
\text{global} & g & ::= & \texttt{Absent} \\
& & \mid & \texttt{FunGlo}\ ((x_{11},\ \ldots,\ x_{n1}).s_1,\ \ldots,\ (x_{1k},\ \ldots,\ x_{nk}).s_k) \\
& & \mid & \texttt{ValGlo}\ ((x_1,\ \ldots,\ x_n).s)
\end{array}
$$

Figure 5.17: The B5 bytecode syntax. Many constructs resemble the CPS language.

Within the code, we refer to globals by integers, so that (for example) `call` expects e_1 (the global) and e_2 (the offset within the bundle) to evaluate to integers. Labels ℓ are just strings. For allocated reference cells, we have a syntactic class of pointers into memory ρ (these are implemented as ML `ref` cells).

The last new element is type and world representations. (These have both expression and value forms; I do not repeat all of the expressions.) The representation of a world $\mathbf{w}$ is always a string ($\mathbf{wrep}$ `"w"`), its address. Many type representations are primitive constants (the syntactic class r). This is because the marshaled format for these types is uniform. For example, a bundle of continuations with CPS type $((\vec{A}_1), \ldots, \vec{A}_n)$ `conts` is always represented as an integer index into the array of globals, no matter what the component types A are. The representation of any `conts` type is therefore $\mathbf{rep}\ \mathbf{Rconts}$. Some other type representations, like the representation of a record type, are still structured. Additionally, some type representations bind representation variables, which are in a different syntactic class γ. We will discuss marshaling and the representations that support it in more detail in Section 5.5.4.

Generation. The translation from a hoisted program to B5 is continuation based. For CPS values we define a function

$$\mathsf{convertv}\ v\ \mathcal{K}$$

that returns a B5 statement. Its arguments are a CPS value v and a meta-level continuation that takes the converted B5 expression and returns a B5 statement. For convenience we also define a function to convert a list of values

$$\mathsf{convertvs}\ (v_1, \ldots, v_n)\ \mathcal{K}$$

where $\mathcal{K}$ takes a list of B5 expressions of the same length. A CPS expression is converted directly to a B5 statement by the function $[\![c]\!]_{\mathsf{B5}}$.

The code generation is a straightforward embedding of the CPS constructs into the B5 ones, erasing types. For example, the **pack** value in CPS is converted into a B5 record containing the type representation and the packed values:

$$\begin{array}{l}\mathsf{convertv}\ (\mathbf{pack}\ A, v_d, v_1, \ldots, v_n\ \mathbf{as}\ \exists\alpha.(B_1, \ldots, B_n))\ \mathcal{K} = \\ \quad \mathsf{convertvs}\ (v_d, v_1, \ldots, v_n)\ \mathcal{K}' \\ \quad \text{where } \mathcal{K}'(e_d, e_1, \ldots, e_n) = \mathcal{K}(\{\mathbf{d} = e_d,\ \mathbf{v1} = e_1,\ \ldots,\ \mathbf{vn} = e_n\})\end{array}$$

Constructs that have only static importance are eliminated entirely. For instance, we compile a $\mathbf{\Lambda}$ with only static (type and world) arguments exactly as its body:

$$\mathsf{convertv}\ (\mathbf{\Lambda}\langle\vec{\alpha};\ \vec{\omega};\ \cdot\rangle.v)\ \mathcal{K} = \ \mathsf{convertv}\ v\ \mathcal{K}$$

We should only see a $\mathbf{\Lambda}$ that takes value arguments at the top-level of a hoisted global, in which case it is compiled to a `ValGlo` with the appropriate arguments. An instantiation of a value of $\forall$ type is complied (when there are value arguments) as

$$\begin{array}{l}\mathsf{convertv}\ (v\ \langle\vec{\alpha};\ \vec{\omega};\ v_1, \ldots, v_n\rangle)\ \mathcal{K} = \\ \quad \mathsf{convertvs}\ (v, v_1, \ldots, v_n)\ \mathcal{K}' \\ \quad \text{where } \mathcal{K}'(e, e_1, \ldots, e_n) = \\ \quad\quad e\ \langle e_1, \ldots, e_n\rangle\end{array}$$

At runtime, we expect e to evaluate to an integer, which we use as an index into the array of hoisted globals. We evaluate that global (which should be a `ValGlo`), which is guaranteed to `return` some value because it is total.

We associate with each CPS variable (valid or modal) a B5 variable, written x_y or x_u:

$$\textsf{convertv}\ y\ \mathcal{K} = \mathcal{K}(x_y)$$

Some CPS values bind variables, which is why the output is a statement:[6]

$$\begin{array}{l}\textsf{convertv}\ (\mathbf{letsham}\ u = v\ \texttt{in}\ v')\ \mathcal{K} = \\ \quad \textsf{convertv}\ v\ \mathcal{K}' \\ \quad \text{where } \mathcal{K}'(e) = \texttt{val}\ x_u = e\ \texttt{in} \\ \qquad\qquad \textsf{convertv}\ v'\ \mathcal{K}\end{array}$$

Finally, we have two ways of interfacing with the local environment. When we see an `extern val`, we presume that we have a corresponding variable available in the initial context:

$$[\![\texttt{extern val}\ y : A@\mathrm{w}' = \ell\ \texttt{in}\ c]\!]_{\textsf{B5}} = (\texttt{val}\ x_y = x_\ell\ \texttt{in}\ [\![c]\!]_{\textsf{B5}})$$

Calls to imported functions are translated to B5's own primitive call:

$$\begin{array}{l}[\![\texttt{let}\ y = \texttt{primcall}(\ell : (A_1, \ldots, A_n) \to B)(v_1, \ldots, v_n)\ \texttt{in}\ c]\!]_{\textsf{B5}} = \\ \quad \textsf{convertvs}\ (v_1,\ \ldots,\ v_n)\ \mathcal{K} \\ \quad \text{where } \mathcal{K}(e_1,\ \ldots,\ e_n) = \\ \qquad \texttt{val}x_y = \texttt{primcall}\ \ell(e_1,\ \ldots,\ e_n)\ \texttt{in} \\ \qquad [\![c]\!]_{\textsf{B5}}\end{array}$$

Execution of B5 is very simple, with the only complications coming from marshaling and the interface with the runtime system. These are discussed in Section 5.5.

JavaScript code generator

The JavaScript code generator is morally the same as the bytecode one, but complicated by the fact that we do not have control over the language. Let's first discuss the fragment of the language that we use.

JavaScript. The JavaScript language (formally known as ECMAScript for the standards body that defines it [32]) is an untyped interpreted language whose basic construct is the object. Objects are property-value associations (hashes). Some specially named properties have special effects, such as the `__proto` property, which indicates a parent "prototype" object which can be consulted in the case that a property is not found in the child object. Some objects are native, representing handles into the DOM

[6]This linearizes the evaluation of values, distorting the nested scopes of these constructs. We know that this will not cause problems because the CPS representation ensures that all variables are globally unique.

$$
\begin{array}{rrcl}
\text{exps} & e & ::= & x \mid \{\ell_1 : e_1, \ldots, \ell_n : e_n\} \\
& & \mid & [e_1, \ldots, e_n] \mid e[e'] \\
& & \mid & e_1 === e_2 \mid e_1 \;\&\&\; e_2 \mid e_1 + e_2 \mid \ldots \\
& & \mid & i \mid \texttt{"string"} \mid \texttt{undefined} \\
& & \mid & e.\ell \mid \ell(e_1, \ldots, e_n) \\
\text{statements} & s & ::= & \texttt{var}\, x = e; s \\
& & \mid & \texttt{switch}(e)\{e_1 : s_1; \texttt{break}; \cdots e_n : s_n; \texttt{break}; \texttt{default} : s; \texttt{break}; \} \\
& & \mid & \texttt{if}(e)\{s_1;\}\texttt{else}\{s_2;\} \\
& & \mid & e; \\
& & \mid & \texttt{return}\, e; \\
& & \mid & \texttt{return};
\end{array}
$$

Figure 5.18: A limited portion of the JavaScript syntax.

tree or events. Manipulating these objects as data, the code takes a simple C- or Java-like imperative form, with familiar `for` and `if` constructs, etc. as well as primitive syntax for creating objects and selecting properties from them.

For the JavaScript code generator we ignore much of the language and use the idealized subset of its syntax shown in Figure 5.18. The expression $\{\ldots\}$ for creating an object uses a colon where we have been using an equals sign throughout the dissertation; it is not a type. The syntax $[e_1, \ldots, e_n]$ creates an array with elements $e_1, \ldots, e_n$, and $e[e']$ subscripts an array. This language is much smaller than B5; we will represent almost everything with objects, using the properties in different ways.

Code generation. As with the B5 code generator, we have continuation-based functions convertv and convertvs for converting values to statements, and $[\![c]\!]_{\text{JS}}$ for converting CPS expressions. The transformation is basically the same, except that we expend extra effort encoding the values of B5 as JavaScript objects. For example, a value of sum type is converted as

$$
\begin{array}{l}
\textsf{convertv}\ (\mathbf{inj}_\ell\ v)\ \mathcal{K} = \\
\qquad \textsf{convertv}\ \mathrm{v}\ \mathcal{K}' \\
\qquad \text{where } \mathcal{K}'(e) = \texttt{var}\, x = \{\mathbf{t} : \ell, \mathbf{v} : e\}; \\
\qquad\qquad\qquad\qquad \mathcal{K}(x)
\end{array}
$$

That is, we represent an injection as an object containing a tag and a value; here the property names $\mathbf{t}$ and $\mathbf{v}$ are fixed strings representing the tag field and value properties respectively. In this sense the code is lower-level than B5 (which had injections as a primitive notion). However, since JavaScript has run-time checks for every field lookup, the code that manipulates these objects is also less efficient. It performs redundant checks (the object will not be `undefined` and the $\mathbf{t}$ property is guaranteed to exist if compilation is correct, etc.) and is needlessly general (the object will only have these two fields, which could be therefore be at known offsets rather than looked up by name).

Many constructs are supported by common code in the runtime system. For example, `newtag` (which creates a new tag for the implementation of extensible types) is translated to a call to the runtime function `lc_newtag`:

$$\begin{array}{l}[\![\texttt{newtag}\ y\ \texttt{of}\ A\ \texttt{in}\ c]\!]_{\text{JS}} = \\ \quad \texttt{val}\ x_y = \texttt{lc\_newtag}()\ \texttt{in} \\ \quad [\![c]\!]_{\text{JS}}\end{array}$$

These runtime functions are implemented by hand in JavaScript. The `lc_newtag` function returns a value $\{\mathbf{a} : \texttt{"addr"}, \mathbf{s} : i\}$ where `"addr"` is the address of the current world and i is an integer that is incremented each time the function is called. We include the generating world's address so that we do not need to coordinate unique integers between hosts, which would be expensive.

We also expand some operations that are primitive in B5. For example, a tagged object is represented as an object with properties $\mathbf{t}$ (the tag) and $\mathbf{v}$ (the embedded value). `untag` is implemented by testing equality of the tag property:

$$\begin{array}{l}[\![\texttt{untag}\ v_o\ \texttt{with}\ v_t\ \texttt{of}\ (\texttt{yes}\ \Rightarrow y.c \mid \texttt{no} \Rightarrow c')]\!]_{\text{JS}} = \\ \quad \textsf{convertvs}\ (v_o, v_t)\ \mathcal{K}' \\ \quad \text{where}\ \mathcal{K}'(e_o, e_t) = \\ \qquad \texttt{if}\ (e_o.\mathbf{t}.\mathbf{a} === e_t.\mathbf{a}\ \ \&\&\ \ e_o.\mathbf{t}.\mathbf{s} === e_t.\mathbf{s})\ \{ \\ \qquad\quad \texttt{var}\ x_y = e_o.\mathbf{v}; \\ \qquad\quad [\![c]\!]_{\text{JS}} \\ \qquad \}\ \texttt{else}\ \{ \\ \qquad\quad [\![c']\!]_{\text{JS}} \\ \qquad \}\end{array}$$

The $===$ operator is "strict equality", which does not attempt to do any conversions (between integers and strings, for example) which would be pointless and dangerous since we know the form of the data we are working on.

For `extern val` we again assume there is a variable (properly, a property of the JavaScript global object) with the same name in scope. For a primitive call, we generate a call to a function of the same name. Currently, all of these values are provided by the runtime system, though the programmer could also write his own JavaScript code and import it this way as well.

Type representations are represented isomorphically to the values in B5, but by using records and strings to distinguish the different cases; the details are uninteresting.

All objects in JavaScript are mutable. This means that we can represent a reference cell as an object with a single property, its contents.

Globals. Code generation produces a declaration of a top-level array `globalcode` containing the globals. Each global is either a hoisted $\mathbf{\Lambda}$ (taking value arguments), a hoisted bundle of continuations, or not present because that global belongs to some different constant world. In the last case, we can put anything we like in that slot; I choose to put a function that displays an error message in case something goes wrong. A hoisted $\mathbf{\Lambda}$ is represented as a JavaScript function taking one argument for each value

argument to the $\mathbf{\Lambda}$. A bundle of continuations is compiled as an array of functions, each taking the same number of arguments as the corresponding continuation.

An instantiation of a value of $\forall$ type is compiled as

$$\begin{array}{l}\textsf{convertv}\ (v\ \langle\vec{\alpha};\ \vec{\omega};\ v_1,\ldots,v_n\rangle)\ \mathcal{K} = \\ \quad \textsf{convertvs}\ (v, v_1, \ldots, v_n)\ \mathcal{K}' \\ \quad \text{where } \mathcal{K}'(e, e_1, \ldots, e_n) = \\ \quad\quad \texttt{var}\ x = \texttt{globalcode}[e](e_1, \ldots, e_n); \\ \quad\quad \mathcal{K}(x)\end{array}$$

where we index into the `globalcode` array and call the function that is there. We expect it to return a value.

As in B5, a function (continuation) is represented as a pair of integers: the index into the global array of code and the offset within that bundle of functions. Thus the **fsel** value is converted as

$$\begin{array}{l}\textsf{convertv}\ (\textbf{fsel}\ v.i)\ \mathcal{K} = \textsf{convertv}\ v\ \mathcal{K}' \\ \quad\quad \text{where } \mathcal{K}'(e) = \\ \quad\quad\quad \texttt{var}\ x = \{\mathbf{g} = e, \mathbf{f} = i\}; \\ \quad\quad\quad \mathcal{K}(x)\end{array}$$

and a call to a function is

$$\begin{array}{l}[\![\texttt{call}\ v_f(v_1, \ldots, v_n)]\!]_{\textsf{JS}} = \\ \quad \textsf{convertvs}\ (v_f, v_1, \ldots, v_n)\ \mathcal{K} \\ \quad \text{where } \mathcal{K}(e_f, e_1, \ldots, e_n) = \\ \quad\quad \texttt{lc\_enqueue}(e_f.\mathbf{g}, e_f.\mathbf{f}, [e_1, \ldots, e_n]); \\ \quad\quad \texttt{return};\end{array}$$

where the runtime function `lc_enqueue` enqueues the parameters (which represent a thread ready to execute) and halts. When the scheduler decides to start this thread, it indexes the `globalcode` array, then the bundle of continuations within that, and calls that function with the supplied arguments.

Say. The `say` M keyword of ML5 produces a string containing a JavaScript expression that, when evaluated, evaluates the ML5 expression M. At this low level, we have `say_cc`, which takes a set of named parameters and a closure-converted continuation.

The named parameters are used to access JavaScript events. This system exists to work around a shortcoming of JavaScript, being that it uses a single global property called `event` to store the current event within an event handler. We must read the properties that we want from this event before the handler returns, or they will be inaccessible. In the CPS construct

$$\texttt{say\_cc}\ x = (\ell_1{:}A_1, \ldots, \ell_m{:}A_m)\ v\ \texttt{in}\ c$$

the label ℓ_i is a string (like `event.keyCode`) that will be interpreted as a JavaScript object selection expression. We expect it to have the type A_i. (In the elaborator we checked that the string belonged to a list of blessed expressions, so only those can appear

here.) The value v is a continuation expression expecting a record of those values as its argument. The translation relies mainly on functionality in the runtime, so is fairly simple:

$$
\begin{array}{l}
[\![\texttt{say\_cc}\ y = (\ell_1{:}A_1, \ldots, \ell_m{:}A_m)\ v\ \texttt{in}\ c]\!]_{\mathsf{JS}} = \\
\quad \mathsf{convertv}\ v\ \mathcal{K} \\
\quad \text{where } \mathcal{K}(e) = \\
\qquad \texttt{var}\ x_1 = \texttt{lc\_saveentry}(e.\mathbf{v1}.\mathbf{g}, e.\mathbf{v1}.\mathbf{f}, e.\mathbf{v0}); \\
\qquad \texttt{var}\ x_y = \texttt{"lc\_runentry("} + x_1 + \texttt{",\{"} + \\
\qquad\qquad \texttt{"l1 : "} + \ell_1 + \cdots + \texttt{", ln : "} + \ell_n + \\
\qquad\qquad \texttt{"\})"}; \\
\qquad [\![c]\!]_{\mathsf{JS}}
\end{array}
$$

The implementation of `say` relies on two runtime functions. The first, `lc_saveentry`, takes a closure without arguments (as the index into the globals, the offset within that bundle, and the environment). We get these from the components of the argument to `say`. This runtime function stores the closure in a table and returns the index into that table, which can be passed to `lc_runentry` to later execute the saved closure on some arguments. The `say` construct returns a JavaScript expression as a string, which is a call to the `lc_runentry` runtime function on the index of the continuation we just saved. It passes along in an object the values of all the event parameters. For example, if we write in our ML5 source program

```
[<input type="text" onkeyup="[say { event.keyCode = c } keypress c]" />]
```

the resulting JavaScript string looks something like

```
'<input type="text" onkeyup="lc_runentry(15, { l1 : event.keyCode })" />'
```

if this is the 15th entry we have placed in the table. Crucially, we read from the global `event` property before returning from the handler. Since `lc_runentry` actually just enqueues a thread and returns immediately, this is our last chance to read from `event` before it is invalidated.

Optimizations. JavaScript is a performance bottleneck in the ML5/pgh implementation, so we perform some optimizations on the output of the code generator. There a few reasons why optimizations are effective here: One is that the type-erasure process has removed a level of abstraction that may have prevented us from recognizing opportunities for simplification. (For example, we can now "see through" what was once **held** or **sham**, since these constructs do not exist in the JavaScript code.) Another is that some type representations have become degenerate (like the representation of `conts` types), which reduces dependencies on other representations, creating more dead code. The third is that because JavaScript is oriented towards using string-based property lookups, we gain a substantial performance increase simply by using short names for variables.

Because it is an imperative, string-oriented language, it is difficult to do optimizations on JavaScript in generality. Therefore, this optimization phase assumes it is running on code that is the output of our code generator. Given that, the optimization

phase is straightforward. It renames variables to be as short as possible, removes dead code, and substitutes small pure expressions.

Output

Once the code generator has produced code for each of the worlds involved in the program, it writes this code to disk—one file per host. This is the end of the compiler's job. Everything else in this chapter is concerned with how these programs are run, which is the responsibility of a collection of software called the runtime system.

5.5 Runtime

We have been looking at details for some time; let us step back and remind ourselves how an ML5 program is executed. From the source program, the compiler produces a series of files—one for each host—containing the part of the program relevant to that host and compiled to its architecture. In order to run the programs, we need a few things: Something must cause the files to be loaded and executed, something must coordinate the communication between hosts, and if the compiled programs need support code, that must be provided as well. These tasks are all the responsibility of the runtime system.

Aside from the fact that we had to make code generators for a concrete set of architectures (B5 and JavaScript), the implementation so far has generally been agnostic about the way that the programs will be run and the way they will communicate. In fact, if we declare other worlds (of kind `javascript` or `bytecode`) in our program, the compiler will happily output code for them. However, the current runtime system is specialized to a two-party computation where the client is running JavaScript and the server is running B5. These are expected to have the names **home** and **server**.

For this runtime, a compiled program consists of two files: the B5 source for the server and the JavaScript source for the client. When the user wants to launch the application, he does so by visiting a URL on a Server 5 web server (Section 5.5.1) that has both compiled files available. It launches a new instance of the application on the server (parsing and loading the B5 code) and sends a tiny web page to the client, which includes the JavaScript runtime code (Section 5.5.3) and the JavaScript output of the compiler. Execution begins on the client, by an effectful expression at the bottom of the JavaScript source file that enqueues the **main** label (now a pair of integers and an array of zero arguments) in the thread queue.

Since this process begins with the web server, let's discuss it first.

5.5.1 Server 5

Server 5 is a small program (about 3500 lines of Standard ML) that listens on a network socket and implements a fragment of the Hypertext Transfer Protocol (HTTP) [38, 39]. Over HTTP it serves static content (such as images used by applications), launches ML5

applications, and tunnels the protocol used by the server and client to communicate within an application instance. It also executes the server half of ML5 applications and provides the interface to server functionality like databases.

Network interface

For simplicity, Server 5 is a single-threaded, single-process program. It is structured around an event loop, which collects all of the outstanding network connections together into a single join point, waiting for activity, and then dispatches events to the modules responsible for those connections.

The networking library constitutes more than one third of the code of the server. It was originally developed for the ConCert project [85]. Its design is somewhat interesting, so I will describe it briefly here. It is based around an abstract type `sock` that represents an outstanding connection:

```
signature NETWORK =
sig
  type sock
  type packet
  (* ... *)

  val send : sock -> packet -> unit
  datatype sockevent =
     Packet of packet * sock
   | Closed of sock
   | Timeout
   | (* ... *)
  val wait : sock list -> time option -> sockevent
end
```

The `send` function sends a packet (described below) on the socket asynchronously—meaning that it is actually queued to be sent when the socket is ready to receive more data. Data can only be received from a socket by using the `wait` function to synchronize on a number of sockets at once (usually all of the sockets in the program). The second parameter to `wait` is a timeout, so that if there is no activity, we will get the `Timeout` event. This is a convenient interface because the network implementation manages the state space of sockets in different modes, for example, a socket that is trying to connect but has not yet succeeded, or a socket with some queued outgoing and incoming messages that is suddenly disconnected. What is interesting about the interface is that each socket is associated with a protocol, which is a way of encoding values of a certain type as bytes to be sent on the network. In the signature:

```
  type 'a protocol

  val protocol :
    { make  : 'a -> string,
```

```
    parse : 'aprefix -> string -> 'a list * 'aprefix,
    empty : 'aprefix } -> 'a protocol

val decode : 'a protocol -> packet -> 'a
val encode : 'a protocol -> 'a -> packet
```

An α `protocol` is a method of converting a value of type α into the abstract `packet` type, and vice versa. We can create a new protocol by explaining how packets are serialized. This consists of four parts. The simplest is a function `make` that turns an α into a string (bytes for the network). When we receive bytes on the network, however, we may not always get enough to constitute an entire α. Therefore, we also need a type $\alpha_{\texttt{prefix}}$ representing some prefix of an α. We always start a new connection with the value `empty`, the prefix represented by no bytes. Finally, when we receive some bytes, we pass those along with the previous prefix to `parse`; the result is zero or more values of type α and a new prefix.

The `socket` and `packet` types are not parameterized because we need to collect all of the sockets together in one list for the event loop. (If they were parameterized by the type of data sent, we would not be able to simultaneously wait on HTTP sockets and `bool` sockets, for example.) The client therefore uses the `encode` and `decode` functions to create packets with some protocol. The `decode` function can fail at runtime if given the wrong protocol for the packet. This is because we use extensible types to implement packets. In the implementation:

```
type packet = exn
type 'a protocol = { encode : 'a -> exn,
                     decode : exn -> 'a,
                     (* ... *) }

fun ('a, 'aprefix) protocol { parse, make, empty } =
  let exception Ptag of 'a
  in
     { encode = Ptag,
       decode = fn (Ptag x) => x
                 | _ => raise Network "bad tag!",
       (* ... *) }
  end
```

The implementation of the α `protocol` type includes a decode and encode function (among other things). When we create a new protocol, we create a new tag for the extensible type `exn` (unfortunately done with the `exception` declaration in Standard ML) which is tested against in the `decode` function. We also need a tag for this protocol's notion of an α prefix in order to build the parser for the protocol, which is not shown here. In practice, a module that wants to use the network declares a single protocol at its head, and uses that same protocol for every connection within the module (meaning that it is very unlikely that we will have a confusion between protocols resulting in a run-time error). The main benefit of this approach is that we do not need to have a

`/5/`*prog*	Create new session of program *prog*
`/toclient/`*id*	Create the server to client socket for session *id*
`/toserver/`*id*	Create the client to server socket for session *id*
`/static/`*file*	Return the file *file* as a web page
`/demos`	Show a list of demos
`/source/`*file*	Display the file *file* as source code

Figure 5.19: The URLs that Server 5 uses to provide access to various functionality. The `toclient` and `toserver` URLs are not accessed directly by users, but by the JavaScript runtime.

centralized datatype enumerating the different kinds of protocols that are used in the application; each module is responsible for implementing its own part of the extensible set of tags itself.

This network interface was particularly useful for ConCert, where we had a number of different packet formats in the same application. By necessity, Server 5 only listens on a single port and uses HTTP for all of its connections. This means that there is currently only a single protocol for the entire application, making this generality somewhat unnecessary. Still, the abstraction is desirable. For example, if we were to add support for more than two hosts in an application, we would need a protocol to communicate among them—HTTP would probably not be appropriate for this.

Event loop

When an HTTP request arrives to Server 5, we decide what to do with it based on the URL that is accessed. If it is static data, we send the requested file from disk. If it communication part of an existing session, we pass it off to that session. If it is a request to launch a new session, we create an instance of the application, a new session identifier, and begin execution. The possible URLs are summarized in Figure 5.19. We begin with the URL for launching an application.

Server runtime

When the URL `/5/`*prog* is accessed by the user, the server loads the B5 and JavaScript files that were the output of the compiler for *prog* (they must have already been compiled). The server parses the B5 code and creates an instance of the application: This just consists of a thread queue (empty, since control begins on the client) and session identifier (an integer). A session also maintains two references to sockets, the "to client" and "to server" sockets. These are used to send messages between the client and server, and are discussed in Section 5.5.2 below. They begin closed. The server then returns a web page to the client that contains the session identifier, the JavaScript code, and the JavaScript runtime. Since the client is where control flow begins, the server then returns to its event loop, with this session idling while it waits for the client to initiate server computation in it.

Abstractly, a message from the client to the server represents a thread of control traveling from client to server via a `go`. (The implementation is discussed below.) When a thread of control arrives, we add it to the thread queue. Interleaved with network activity, Server 5 is also running a simple interpreter for B5. For each session, we check to see if it has any outstanding threads that can be run. If so, we run some. (The scheduling policy is round-robin and tuned for responsiveness rather than throughput.) A thread can only run for a finite time before yielding because of the way that we have compiled programs: Recursion is the only way for a program to loop, but `call`s are compiled to push threads into the thread queue rather than looping eagerly. Therefore we do not need to worry about preemption; we run the threads to completion and go on to the next round. Additionally, if there are any queued messages from the server to the client and we are in a position to send them, we do so.

Server 5 also provides some functionality that programs can access directly, such as a rudimentary database (Section 5.1.4). The only interesting part of this is the implementation of `addhook`. Recall that it is imported to ML5 as

```
extern val trivialdb.addhook
    : string * unit cont -> unit @ server
```

This allows us to register a thread that will be executed whenever the indicated key is updated. The implementation of the database just associates a list of such hooks with each key, and when a key is updated, the continuation is launched in the same way as a message from the client causes a continuation to be run. Because the database is shared by all of the application instances running on Server 5, this gives us a limited (but usually sufficient) way of having different application instances interact. We will see several examples of this in Chapter 6. Of course, the right way to support applications with more than two hosts would be to use the expressive language that we have gone through so much trouble to build. The ability to do so is limited by runtime support, with the chief limitation being the JavaScript security model. I describe this and the way that we work around it in the next section.

5.5.2 Communication

The JavaScript security model is based on traditional sandboxing techniques (for preserving memory safety on the client machine) and the *same origin policy* for enforcing access control. The same origin policy asserts that JavaScript code can only access things (DOM nodes, network sockets, etc.) that are from the same origin, where the "origin" is taken to be the host, port, and URL that served the JavaScript [61, 120]. Furthermore, the only way that JavaScript is able to communicate with other hosts is by actively opening a connection;[7] it is not possible for it to open a listening socket and wait for connections from other hosts.

[7]There are in fact many ways of doing this to work around various browser quirks and attempts at patching security holes. These range from the straightforward one that we use here to dynamically generating HTML content that includes tiny invisible images (whose loading creates a minute covert channel with the server that hosts them).

This causes trouble for the ML5 model, where the server can initiate a thread on the client with `go` at any time. The solution is standard in web programming practice: The client preemptively makes a request to the server that the server delays responding to until it wants to send a message. When it sends a message, the client destroys the connection and creates a new one to replace it.

To make this work, the server keeps a queue of messages that it wants to send to the client. The client attempts to always keep open a connection to the server by fetching the `toclient` URL whenever it does not have an outstanding request. This has the nice side effect of serving as the application's keep-alive; even if the application is idle and not making requests of the server, it still must create this connection. If it does not, then the server can assume that the client has left the page and destroy the session to reclaim resources.

This technique is reasonably simple and works well, because we always make a connection back to the server that originally provided the JavaScript code. Unfortunately, allowing for connections between clients or to other servers is much more difficult because of the same origin policy. The most general way to support multiple hosts would be to build an overlay network where messages were routed through the origin server(s) in a way that respected the same origin policy. This would be a nice next step for Server 5.

5.5.3 Client runtime

The implementation of the client runtime is simpler in many ways, because the web browser already provides a lot of the functionality that we need.

Like the server, the client has a thread queue and a simple round-robin scheduler. The web browser runs JavaScript in a single thread—halting the user interface until it is complete—so ML5/pgh's CPS-based compilation and explicit yielding are necessary here. (Many advanced web programmers manually CPS- and closure-convert their programs for this reason, in fact.) Every `call` was compiled as a call to the runtime function `lc_enqueue`. The simplest implementation of it looks like this:

```
function lc_enqueue(g, f, args) {
  lc_threadqueue.enq( { g : g, f : f, args : args } );
  setTimeout(lc_schedule, 10);
};
```

We just add the continuation and its arguments to the bottom of the thread queue, and then set a timer to run the scheduler once after 10 milliseconds. We don't achieve 10ms scheduling frequency in practice; this low number just gives the browser a chance to process other events before returning to the ML5/pgh scheduler. In fact, it turns out that the scheduling frequency is so low that we get abysmal performance if we yield at every function call. An improved implementation of `lc_enqueue` is therefore:

```
var THREADPACE = 6;
var lc_recsleft = 0;
function lc_enqueue(g, f, args) {
```

```
  if (lc_recsleft === 0) {
    lc_threadqueue.enq( { g : g, f : f, args : args } );
    setTimeout(lc_schedule, 10);
    lc_recsleft = THREADPACE;
  } else {
    lc_recsleft --;
    var f = globalcode[g][f];
    f.apply(undefined, args);
  }
};
```

This version eagerly continues execution for 6 steps before yielding, by decrementing a counter and then immediately scheduling the thread. This is a tradeoff between the responsiveness of the user interface and the throughput of the JavaScript computations. We can not set it very high before being troubled by recursion limits or making the application behave badly, but small factors give an almost linear speedup. A better implementation would have a finer cost model for the underlying code [136]; right now we only measure the cost in terms of the number of calls. The cost of each function can vary widely depending on the program and how it happened to be compiled.

We arrange that we only run the scheduler when there are threads in it, so we never waste time repeatedly checking an empty queue. We do this by installing a timer only when we have added a thread to the queue, so that there is exactly one outstanding JavaScript timer for each ML5 thread in the queue. This means that the thread queue only consists of threads that are ready to run.

There are other sources of "waiting" threads, however, such as event handlers in the UI and messages from the server. These become active through mechanisms other than the thread queue. For example, JavaScript provides the `XMLHttpRequest` object, which is used to fetch the `toserver` and `toclient` URLs and establish communication with the server. The interface to XMLHttpRequest is asynchronous, meaning that we give it a handler that runs whenever it receives data from the server. The handler for our `toclient` connection checks to see if it has read a complete marshaled message (which consists of a closure to execute); if so, it unmarshals the closure and puts it in the thread queue. It then makes a replacement `toclient` connection to wait for further messages. Event handlers that we placed on UI elements using `say` can also enqueue threads; these come from the table of continuations that `say` saves into.

The client runtime also provides the implementation of some functionality that we import using `extern val`. Though the interface we give for the DOM is very low-level, it nonetheless requires brief stubs to implement it, since DOM operations are typically properties of the objects rather than "static" methods. For example, the function that changes the UI focus to a DOM element is imported in ML5 as

```
extern val dom.focus : dom.node -> unit @ home = lc_domfocus
```

and implemented in the runtime as

```
function lc_domfocus(node) {
    node.focus ();
```

```
};
```

Most of the JavaScript runtime consists of the marshaling and unmarshaling routines, which are the subject of the next section.

5.5.4 Marshaling and unmarshaling

Marshaling is the process of transforming an arbitrary runtime value into a string that can be interpreted by another host to reproduce the value by the inverse process, unmarshaling. Both marshaling and unmarshaling take as an argument the run-time representation of the type of the value in question.

In ML5, marshaling and unmarshaling use the same algorithm on all of the hosts, because they must agree on the format of marshaled data. Therefore I will discuss marshaling from the server's perspective, without loss of generality.

Marshaling. The marshal function takes the value and a representation of its type and returns a string. It also takes a marshaling context Δ, which maps variables γ to type and world representation values. We need this because some representations have binding structure. Finally, marshal keeps track of an optional concrete world $\mathbf{w}^?$, which is the world of the value being marshaled. The marshaling context begins empty and the world begins as the world doing the marshaling.

Most of the cases of marshaling are very simple, and just consist of converting the value to a string in an arbitrary but consistent way. Rather than bother with the details of how these strings are actually represented, I pretend as though marshaling produces a list of tokens, which can be strings, integers, or other lists of tokens. For example, the case for marshaling tags is as follows:

$$\mathsf{marshal}\ \Delta\ (\mathbf{tag}\ \texttt{"addr"}, i)\ (\mathbf{rep}\ \mathbf{Rtag})\ \mathbf{w}^? \quad = \quad [\texttt{"addr"}, i]$$

That is, to marshal a tag (a value consisting of the address of the host that created the tag, as a string, and a unique integer), we produce the address and integer. A less degenerate case is for records:

$$\begin{aligned}\mathsf{marshal}\ \Delta\ (\{\!|\ell_1 = v_1, \ldots, \ell_n = v_n|\!\})\ (\mathbf{rep}\ \{\ell_1 : v_{r1}, \ldots, \ell_n : v_{rn}\})\ \mathbf{w}^? = \\ [\ell_1, \mathsf{marshal}\ \Delta\ v_1\ v_{r1}\ \mathbf{w}^?,\ \ldots,\ \ell_n, \mathsf{marshal}\ \Delta\ v_n\ v_{rn}\ \mathbf{w}^?]\end{aligned}$$

In this case we just recursively marshal each of the component values along with the name of its label.[8] The marshaling of integers and strings, injections, continuations (which are pairs of integers), addresses, type and world representations themselves, and most other values are just straightforward recursive representations of their syntax. I do not give those here, instead discussing only the remaining cases that are interesting.

The marshaling function is defined by case analysis on the type representation, not the value being marshaled. For example, the **held** constructor is present at run time, but

[8]We could be more efficient by insisting that the values be sorted by name, or by not using named labels at all, but that is an optimization for a later time.

we do represent the `at` type:

$$\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}(v_t\ \mathtt{at}\ v_w))\ \mathbf{w}^? = \mathsf{marshal}\ \Delta\ v\ v_t\ (\mathsf{read}\ (\Delta, v_w))$$

Here we recursively marshal the same value at the representation v_t. We also change the current world, by using the read function, which is defined as

$$\begin{array}{lcl} \mathsf{read}\ (\Delta, \mathbf{wrep}\ \mathtt{"w"}) & = & \mathbf{w} \\ \mathsf{read}\ (\Delta, \mathbf{rep}\ \gamma) & = & \Delta(\gamma) \end{array}$$

The world may be either a literal world representation (in which case we extract it), or it may be a representation variable, in which case we look it up in the context. In the context it may have the value "—," the reason for such optional worlds are discussed in the case for Ξ below.

The reason that we record the current world is so that we can specialize the marshaling of certain values—specifically local resources—based on the world that we are marshaling them at. Suppose that we are marshaling a local mutable reference cell.

$$\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}\ \mathbf{Rref})\ \mathbf{w}^? = \ldots$$

We use the primitive **Rref** representation for mutable references and other local resources, so the value v could have the form $\mathbf{ptr}\ \rho$, for example. Even if we knew that it had this form, we cannot directly marshal pointer into memory ρ. This is because we cannot get access to this pointer in JavaScript or ML, nor would we want to send raw pointers over the network—it would interfere with garbage collection, for instance. What we do instead is called *desiccation*: we store v in a local table and marshal the integer index into that table instead. We only want to do this if the reference is a local reference. The representation **Rref** does not tell us whether it is a local or remote reference, so this is where we use the world parameter to marshal. Supposing that we are marshaling on the server, then the case is

$$\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}\ \mathbf{Rref})\ \mathbf{server} = [i]$$

where i is the index of v after it has been inserted into our local table. If the world does not match, then the case is

$$\mathsf{marshal}\ \Delta\ i\ (\mathbf{rep}\ \mathbf{Rref})\ \mathbf{w}' = [i] \qquad (\text{when } \mathbf{w}' \neq \mathbf{server})$$

In this case, we are marshaling a local reference that belongs to some other world. If we have such a reference, then it was desiccated before it was sent to us, so it is locally represented as an integer. Therefore, we expect an integer and marshal it as an integer. Reconstitution of desiccated values takes place in unmarshaling, which is described below.

The representation might be a variable, in which case we look it up in the context:

$$\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}\ \gamma)\ \mathbf{w}^? = \mathsf{marshal}\ \Delta\ v\ (\Delta(\gamma))\ \mathbf{w}^?$$

Note that this rule applies for any value v. We bind representation variables as we recurse. For example, inductive types are marshaled as follows:

$$\begin{aligned}\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}\ (\pi_i(\mu(\gamma_1.v_1,\ \ldots,\ \gamma_n.v_n))))\ \mathbf{w}^? = \\ \mathsf{marshal}\ (\Delta, \gamma_1 = r(1), \ldots, \gamma_n = r(n))\ v\ v_i\ \mathbf{w}^? \\ \text{where } r(k) = (\mathbf{rep}\ (\pi_k(\mu(\gamma_1.v_1,\ \ldots,\ \gamma_n.v_n))))\end{aligned}$$

Because the **roll** coercion is erased at runtime, the value could take any form (in practice it is an **inj**, because these come from source language datatypes). We will continue marshaling the same value at an unrolled version of the representation. We extend the context to bind the representation variables γ; each one is bound to the representation of the corresponding projection from the μ. This case of marshaling does not make the representation smaller (by some measure), so it is how we are able to marshal arbitrarily large objects without necessarily having representations of the same size.

Three other constructs bind variables during marshaling. First, a value of existential type is marshaled as follows:

$$\begin{aligned}\mathsf{marshal}\ \Delta\ (\{\!\{\mathbf{d} = v_r, \mathbf{v1} = v_1, \ldots, \mathbf{vn} = v_n\}\!\})\ (\mathbf{rep}\ (\exists\gamma.v_{r1},\ \ldots,\ v_{rn}))\ \mathbf{w}^? = \\ [\mathsf{marshal}\ \Delta\ v_r\ (\mathbf{rep}\ \mathbf{Rrep})\ \mathbf{w}^?, \\ \mathsf{marshal}\ (\Delta, \gamma = v_r)\ v_1\ v_{r1}\ \mathbf{w}^?, \\ \vdots \\ \mathsf{marshal}\ (\Delta, \gamma = v_r)\ v_n\ v_{rn}\ \mathbf{w}^?]\end{aligned}$$

The representation of $\exists\alpha.\vec{A}$ binds a variable to the representation of α. This comes from the **d** field of the value that has that type—since the hidden type could be many different things, we can not know it ahead of time. We marshal it and send it, since the unmarshaling side will need to get the representation dynamically as well; when marshaling the representation we use the representation of representations, **rep Rrep**. This is followed by the marshaled values, which are marshaled in the extended marshaling context.

Marshaling of $\forall$ and $\Box$ types is subtle because they are universal quantifiers, not existential ones. This means that there is no one existential witness to bind to the representation variable. Because the quantifier is universal, however, we can instead choose any instantiation of it that we like. Let's take a $\forall$ quantifier that abstracts only over types as an example:

$$\begin{aligned}\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}(\forall\langle\gamma_1,\ \ldots,\ \gamma_n;\cdot;\cdot\rangle.v_r))\ \mathbf{w}^? = \\ \mathsf{marshal}\ (\Delta, \gamma_1 = \mathbf{rep}\ \mathbf{Rvoid}, \ldots, \gamma_n = \mathbf{rep}\ \mathbf{Rvoid})\ v\ v_r\ \mathbf{w}^?\end{aligned}$$

We choose to instantiate the universal type at the empty type `void`, and therefore bind all of the representation variables to **rep Rvoid**. Marshaling at this representation always fails, but there are no values of `void` type! Similarly, there is no value of type $\forall\langle\alpha;\cdot;\cdot\rangle.\alpha$ (or else there would be a value of type `void`). On the other hand, we do have values of type $\forall\langle\alpha;\cdot;\cdot\rangle.(\alpha\ \texttt{cont})$. The marshaling of a continuation does not depend on the types of its arguments, so instantiating α with `void` here does not cause trouble (we

never look up the corresponding representation variable γ). As another example, the sum type $[\ell_1 : \texttt{int}, \ell_2 : \texttt{void}]$ is inhabited, but only by a value of the form $\mathbf{inj}_{\ell_1}\ n$. This means that we can marshal the compiled version of the polymorphic empty list **nil**, because the arm of the sum type that it occupies does not mention the polymorphic type variable.

The fact that we can choose to instantiate a universal quantifier any way we want is our justification for marshaling $\forall$ this way. That we never try to marshal at the **Rvoid** representation is a consequence of a correct implementation. In fact, since a correctly compiled program should never use the representation variables bound by $\forall$, an optimized implementation should probably not represent the $\forall$ type at all. Either way, it is important to observe that this is what we are doing.

The $\mathfrak{B}_\omega$ type constructor is also a universal quantifier. We treat it in the same spirit as $\forall$, but have no direct analog of the `void` type for worlds. This is the reason that the world argument to marshal is optional:

$$\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}\ (\mathfrak{B}_\gamma.v_r))\ \mathbf{w}^? = \mathsf{marshal}\ (\Delta, \gamma = —)\ v\ v_r\ —$$

When we marshal a value of $\mathfrak{B}$ type, we bind the representation variable to "—" and also begin marshaling at no concrete world. This is slightly different than the case for `void` in that there are many values that are well-formed irrespective of the world (the cases for records and integers do not care about the world at all, for example). The only marshaling cases that are sensitive to the world are the ones that desiccate local resources. There are no local resource values that are well-formed at every world, however, so these cases do not arise. Worlds can also appear in some types. If this universally-quantified world appears in an `at` type, we will just set the current world to "—" like we did with $\mathfrak{B}$. They can appear in some other values, like the type of the address of a world. Again, there is no value of address type that works for all worlds; every address is for a specific world. Therefore, we should not encounter these cases either. Like $\forall$, we might as well not even represent the $\mathfrak{B}$ constructor—but it is worthwhile to observe our justification for doing so.

For the same reasons, in the full version of the rule for $\forall$, we bind the representation variables for the world arguments to "—"; the case is

$$\begin{aligned}&\mathsf{marshal}\ \Delta\ v\ (\mathbf{rep}(\forall\langle\gamma_1,\ \ldots,\ \gamma_n;\gamma'_1,\ldots,\gamma'_m;\cdot\rangle.v_r))\ \mathbf{w}^? = \\ &\qquad \mathsf{marshal}\ (\Delta, \gamma_1 = \mathbf{rep}\ \mathbf{Rvoid}, \ldots, \gamma_n = \mathbf{rep}\ \mathbf{Rvoid}, \gamma'_1 = —, \ldots, \gamma'_m = —)\ v\ v_r\ \mathbf{w}^?\end{aligned}$$

Unmarshaling. Unmarshaling is the inverse of marshaling. It is a function

$$\mathsf{unmarshal}\ l\ v_r\ \mathbf{w}^?$$

where l is a list of tokens, v_r is the representation of the type of value we expect, and $\mathbf{w}^?$ is an optional concrete world as in marshaling. It returns the unmarshaled value and some suffix of l. We write $l_1 + l_2$ for the concatenation of token lists l_1 and l_2.

At the server we always begin unmarshaling at the same type,

$$\exists\alpha.(\alpha, \alpha\ \texttt{cont}))\ \texttt{at}\ \mathbf{server}$$

When we finish unmarshaling, we have a local continuation and an argument for it, so we can insert this in our local thread queue.

Unmarshaling is simple in most cases, reversing the work done by marshaling. For example, to unmarshal a value of sum type (when the arm carries a value):

$$\begin{array}{l}\mathsf{unmarshal}\ ([\ell]+l)\ (\mathbf{rep}\ [\ell : v_r, \vdots])\ \mathbf{w}^? = \\ \qquad (\mathbf{inj}_\ell\ v, l') \\ \qquad \text{when } \mathsf{unmarshal}\ l\ v_r\ \mathbf{w}^? = (v, l')\end{array}$$

Unmarshaling treats the `at` modality, the universal quantifiers, and inductive types the same way that marshaling does. For existential types,

$$\begin{array}{l}\mathsf{unmarshal}\ \Delta\ l\ (\mathbf{rep}\ (\exists\gamma.v_{r1},\ \ldots,\ v_{rn}))\ \mathbf{w}^? = \\ \qquad (\{\!\{\mathbf{d} = v_r, \mathbf{v1} = v_1, \ldots, \mathbf{vn} = v_n\}\!\}, l') \\ \qquad \text{when} \quad \mathsf{unmarshal}\ \Delta\ l\ (\mathbf{rep}\ \mathbf{Rrep})\ \mathbf{w}^? = (v_r, l_1) \\ \qquad \text{and} \quad \mathsf{unmarshal}\ (\Delta, \gamma = v_r)\ l_1\ v_{r1}\ \mathbf{w}^? = (v_1, l_2) \\ \qquad\qquad \vdots \\ \qquad \text{and} \quad \mathsf{unmarshal}\ (\Delta, \gamma = v_r)\ l_n\ v_{rn}\ \mathbf{w}^? = (v_n, l')\end{array}$$

We begin by reading the representation of the hidden type from the list of tokens, using the representation of the type of representations, (**rep Rrep**). We then bind that to the representation variable and read the sequence of values using the sequence of representations that are components of the $\exists$.

The inverse of desiccation is *reconstitution*. When unmarshaling a reference, we again have two cases depending on the world:

$$\mathsf{unmarshal}\ \Delta\ ([i]+l)\ (\mathbf{rep}\ \mathbf{Rref})\ \mathbf{w}' = (i, l) \qquad (\text{when } \mathbf{w}' \neq \mathbf{server})$$

The marshaled format of a reference is always a desiccated integer; if it is not our own, then we continue to represent it as an integer. If it is at **server**, however, we read the value from our local table:

$$\mathsf{unmarshal}\ \Delta\ ([i]+l)\ (\mathbf{rep}\ \mathbf{Rref})\ \mathbf{server} = (v, l)$$

where v is the value in the local table at position i.

Other considerations

Sharing and cycles. Marshaling does not preserve sharing (where two identical objects live are allocated to the same place in memory). Sharing is important for some functional algorithms in order to get good space behavior. In the Grid/ML marshaling implementation we used pointer comparisons to preserve sharing and cycles [85]. We do not have direct access to pointer values in JavaScript, which makes sharing-preserving marshaling hard to implement. Our best strategy would probably be hash consing [6, 36], which would also have the advantage of finding accidental sharing in

addition to the sharing already present, but would be quite costly. On the other hand, cycles pose no problem because they can only come from mutable references. We don't look at the contents of mutable references, because we marshal them by desiccation. In any case, the design of the language does not constrain our implementation of marshaling much: only that we need to be able to marshal any value. Since desiccation is a fairly general way of handling local resources like reference cells and DOM handles, the remaining data can be marshaled using whatever technology is appropriate.

Garbage. One problem with desiccation, however, is that it accumulates values in the local table. Currently, these can not be reclaimed. To do so, we would have to implement distributed garbage collection, because the last outstanding reference to a desiccated value might be as an index in a data structure at another world. Distributed garbage collection is possible in principle, but quite difficult. Fortunately, we only pay this cost when we use certain mutable structures and local resources—the local garbage collectors in the web browser and Standard ML runtime can collect anything else. I do not think that these unreclaimable resources are a big problem for our application domain. Instances of web programs are usually short-lived, and users tolerate web browsers that leak memory for existing web applications. An implementation that additionally allowed for long-running programs would clearly be useful, however.

Data format. The server and client communicate using marshaled strings in the bodies of HTTP requests. One good thing about using a mainstream protocol like HTTP for communication is that network software usually assumes that the traffic is desirable and therefore allows it through proxies. Unfortunately, some software also assumes that it understands the contents of the request, and therefore will modify it! (For example, some VPN software will rewrite the page to insert a control panel and modify links to pass through the VPN.) I found it necessary to wrap the marshaled message in a tag `<ml5>`s`</ml5>` and use a custom MIME type [41] in order to work around proxies; it is not clear how general this solution is, but hard to accept blame for it not working.

5.6 Summary

In this chapter I described the programming language ML5, which is an ML-like language for distributed computing based on the modal typing discipline that we derived from the modal logic IS5 (Chapter 3). The language retains the spirit of ML: it is statically typed, higher-order, call-by-value, and has support for both functional and effectful programming. Our modal typing judgment integrates cleanly into the language, leading to an extension of the usual polymorphic type inference algorithm to type, world, and validity inference. Owing in part to type inference, SML code in the common subset generally type-checks without modification. In this sense the distributed features and typing discipline do not interfere with programs that do not make use of them.

I also described the type-directed ML5/pgh compiler, which follows the same basic strategy as the abstract compilation I formalized in the previous chapter. It is extended so that we can implement first-class continuations, algebraic datatypes and extensible types, type polymorphism, an interface to local resources via network signatures, runtime type representations for marshaling, and exceptions. The compiler is supported by a runtime system, including a web server and interpreter for the server's bytecode language, as well as implementations of the marshaling algorithm for both client and server.

Implementing a language is an intellectually valuable exercise even if we never use it, but it is clearly also interesting to try to write programs in our language! The implementation currently works well enough to support some realistic demo web applications. These are the subject of the next chapter.

Chapter 6

Applications

In this chapter I describe some of the applications that we have built using ML5. These applications are intended as demonstrations and exercises of its features, so they are somewhat small. Nonetheless, they suggest that with some more engineering, ML5 could be used to build full-scale applications, and show us what features work well and which ones need to be improved.

The screenshots in this chapter have been modified slightly to make them more appropriate for print. The applications can also be run online at `http://tom7.org/ml5/` or by installing and running Server 5 on a local computer.

6.1 Watchkey

In developing these applications I tried not to implement too much functionality on the server in Standard ML or the client in JavaScript. This is because I didn't want ML5 code to just be "glue" between code written in native languages (although it can clearly be used this way)—most of the computation should be expressed in ML5. One consequence is that the database that all of the applications use is very rudimentary. It is imported on the server with the following network signature:

```
extern val trivialdb.read : string -> string @ server
extern val trivialdb.update : string * string -> unit @ server
extern val trivialdb.addhook
    : string * unit cont -> unit @ server
```

It associates string keys with string values. We use this simple database for persistent storage in all of the applications. Let's look at a simple use of it before moving on to the real applications.

The *Watchkey* demo displays the current value of a database key on the server, and allows the user to modify that key. A screenshot appears in Figure 6.1. It is good to see one working program in its entirety; the source code is as follows:

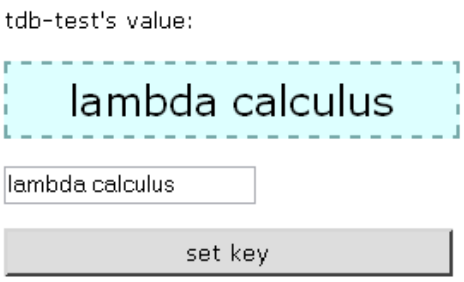

Figure 6.1: A screenshot of the *Watchkey* demo. The upper box (with DOM id `showbox`) displays the current value of the database key; it is asynchronously updated whenever the key is changed on the server. The input form on the bottom (with DOM id `inbox`) allows the user to set the key by pressing the button.

```
unit
  import "std.mlh"
  import "dom.mlh"
  import "trivialdb.mlh"
  put k = [tdb-test]
  fun getkey () =
      let val v = from server get trivialdb.read k
      in  dom.setstring (dom.getbyid [showbox], [innerHTML], v) end
  fun setkey () =
      let put s = dom.getstring (dom.getbyid [inbox], [value])
      in  from server get trivialdb.update (k, s) end
  do dom.setstring (dom.getbyid [page], [innerHTML],
                    [[k]'s value:
                     <div class="show" id="showbox"> </div> <br />
                     <input class="in" type="text" id="inbox" /> <br />
                     <div onclick="[say setkey ()]"
                          class="button">set key</div> ])
  do from server
     get trivialdb.addhook (k, cont (fn () => from home get getkey ()))
end
```

We begin by importing header files that define the standard environment and import the external resources for interacting with the client DOM and server database. The database key that we use in this application is `tdb-test`, which is bound to the valid variable `k`. We use `k` on both the client and server below. (Note that we do not have to use `put` here to make the binding of `k` valid—the binding would be generalized because the right hand side is a value.) The function `getkey` retrieves the value of the key from the server and modifies the page to display it. It uses `get` to evaluate a database read expression on the server, and then uses the DOM routines to modify the page on the client when it returns. The display box (which is later created as part of the page) will have the DOM id `showbox`; setting its `innerHTML` property changes its displayed contents. The `setkey` function reads the value of the input box (with id `inbox`), binds

it to a valid variable `s` so that we can use it on the server, and then goes to the server to update the database. Both of these functions have type `unit` → `unit@`**home**; they are modal because they access the DOM tree, which is local to the client.

The next declaration has the effect of setting the entire page to the HTML document given. The document contains the HTML elements that `getkey` and `setkey` refer to. It also contains an element that handles the `onclick` event by calling `setkey`. The CSS classes [11, 69] `show`, `in`, and `button` just change the presentation of the elements so that (for example) the clickable button has a raised outline. The declaration of these styles are not important to the behavior of the program and so I don't show them.

After creating the page, we also make a trip to the server to register a hook for when the key changes. We use the `addhook` function to do this. The hook that we send goes `home` and runs the `getkey` function to update the contents of the display.[1] The `cont` function is defined in the standard library; it converts a function of type `unit` → `unit` and to a `unit cont`.

That's it! The remainder of the applications use a similar model of creating a web page with event handlers and modifying its contents in response to activity from the user or asynchronous updates from the server. Since they are much larger, we will not want to look at the entire source code. Instead, we will discuss their design, the most interesting parts of their implementation, and how the features (or lack of features) in ML5 help and hinder us.

6.2 Chat

One consequence of using a persistent centralized database is that multiple instances of the same application see the effects of each other's database modifications. For example, two instances of the *Watchkey* demo can see each other's updates to the key. The first application is a generalization of this mode of use to a two-party chat program.

The design of this application is based on having a database key for each of the two players, representing their current message. (We actually have two keys per player, the "old message" and the "current message".) These have hooks installed, like in *Watchkey*, so that the two players see updates to the keys as they are made. There are two phases of the application. When it starts, the user selects which of the two players he is, and then it proceeds to the actual chat.

Selecting the player consists of displaying a web page with two buttons on it. This screen appears in Figure 6.2. First class continuations give us an elegant way to display this prompt and react to the selection of a player. The beginning of the program looks like this:

```
put (us, them) =
  letcc ret
```

[1]This program can be made more efficient by reading the key before it goes to the client, so that it does not need to make another round trip. This optimization could plausibly be done by the compiler, in fact, because the `get` obeys certain equivalences. Such an optimization would be useful and a nice consequence of using high-level language features to express the concepts of distribution.

Figure 6.2: Choosing the player in the chat application. Each of the buttons has an event-handler that continues at the top-level of the program with bindings for the player selected.

```
in
 let fun pick p = throw p to ret
 in
  dom.setstring
  (dom.getbyid [page],
   [innerHTML],
   [<div class="heading">Chat! Choose your player:</div> <br />
    <span class="button"
     onclick="[say pick ([chat.1], [chat.2])]">player 1</span>
    <span class="button"
     onclick="[say pick ([chat.2], [chat.1])]">player 2</span>
    ]);
   halt ()
 end
end
```

We are making a declaration of the pair of variables *us* and `them`, which are the database keys for this player and the opposite player. (If we choose player 2, then `us` is `chat.2` and `them` is `chat.1`). We can't wait in a loop for the prompt to return, because this would lock the user out of the interface; instead we must react to actions initiated by the user. To do so, we save the current continuation (of type (`string` × `string`) `cont@home`). The function `pick` takes a pair of strings and throws them to the return continuation that we just saved. We then modify the page to display the prompt. It has two buttons with event handlers. The "player 1" button calls `pick` with the keys for the first player, and similarly for player 2. We then terminate the current (top-level!) thread by calling the standard library function `halt`. The page waits idle until the user triggers one of the two event handlers, at which point the appropriate strings are bound by the `put` and we continue to the later top-level declarations, which enter the chat mode.

One shortcoming of this approach, which is a problem with all of our applications, is race conditions. It is conceivable that (depending on how the program is compiled and scheduled) the user could click both buttons and thus launch two concurrent threads running the remainder of the application. This would not cause the application to crash, but it would behave strangely (two competing server hooks would be added for each of the keys, for example). There are a number of solutions we could contemplate for this. It would be easy to add rudimentary locking or atomicity to the language because JavaScript execution is already single-threaded. The intention that an event handler (or an external disjunction of event handlers) be *linear* is common in these applications—that it (and related alternatives) should only be used once. This would also be simple to implement because JavaScript is single-threaded. In any case, a proper treatment of user-interface concurrency is outside the scope of this dissertation, and so I do not worry about it much in these applications.

After selecting the `us` and `them` keys, we proceed to replacing the page with the chat interface. This appears in Figure 6.3. The chat bubbles are just styled `<div>` elements, whose bodies we update as in the *Watchkey* demo. They show the value of the "old" key (in grey) and the current key (in black). The input form at the bottom is the only source of user input. It has an event handler registered for keypresses:

```
[<input type="text"
   class="typebox"
   onkeyup="[say { event.keyCode = c }
             case c of
                 ?\r => input-send ()
               | _ => send-all ()]"
   id="[id.input]" />]
```

The `send-all` function reads the input box and sends its contents to the server to be saved at the `us` key. Because the other player will have a hook attached to this key, he will see our message as we type it. If the return key is pressed (the keyCode is the character `\r`) then we call the `input-send` function. This replaces the "old" `us` key with the value of the current one, and blanks the current key, to visually indicate that the player has committed to his message. This was the impetus for adding event parameters to the `say` construct; without them there is no way to detect that the user has pressed the return key.

That's all there is to this application. There are a couple things that could be improved about it. First, the application could automatically assign players to slots (or support an arbitrary number of concurrent players). To do so it would have to use the database to arrange that each player is assigned a unique identifier and key, and that the list of active players is sent to each of them. The database is not really the most appropriate structure for doing this. Moreover, since ML5 is already designed to support multiple hosts in the same computation, it would be better to build the application that way and improve the runtime to support multiple hosts in an application. I discuss some ways this could possibly be accomplished in Chapter 7. Another problem is that the application sends a lot of redundant messages. For each keypress, we make a

Figure 6.3: Chatting in the chat application. The bottom bubble is the user's chat, and the top is the opposite player's. A form tracks the user's input as he types.

round-trip to the server and send not just that keypress but the entire string. Moreover, the thread model does not guarantee that these threads will run in order, so an earlier keypress could overwrite a later one. The cost of sending the entire input each time is not serious, since the fixed overhead of a message is the overwhelming factor for the input sizes we're concerned with here. Sending too many messages and the possibility for sending them out of order is a more serious problem, however. In the next application we will see one way of avoiding this problem.

6.3 Wiki

The next application is a Wiki, which is a collection of hyperlinked documents that are easily editable by the user. The idea of this application is to associate each document with a database key, and provide an interface for navigating between keys and editing them. Its most important feature is that each document is rendered using a custom markup language. All of this rendering is performed on the client.

The interface and markup language are designed to mimic the popular MediaWiki software [71], which is used for Wikipedia [143]. Unlike MediaWiki, navigation between documents is performed all within the same web page (rather than using the browser's native hyperlinks). To begin editing a page, the user clicks on the "edit" tab at the top, which displays the editable document source without leaving the page. A screenshot of the editing interface appears in Figure 6.4.

This application would be just like the *Watchkey* demo if not for the client-side page rendering. As the user modifies the page source, a thread parses the markup language and generates the rendered HTML for the display. The syntax `[[dest|text]]` creates a link to the page `dest` that displays with the text `text`. These links are not real hyperlinks but colored text with an `onclick` handler generated by `say` that navigates to the referenced page. The wiki also supports the syntax `{{tmpl|s1|arg=s2|arg2=s3|s4}}` for templates. A template is a parameterized page fragment that can be used in the construction of other pages. The arguments are separated by a vertical bar, and can be named or unnamed. (For example, the `heading` template used in the screenshot renders its unnamed argument at a larger size followed by a horizontal rule.) Within the definition of the template (itself just a page), uses of the arguments are written `{{{name}}}` for a named argument or {{{n}}} for the n^{th} unnamed argument. Rendering a page therefore may require the contents of other pages. Additionally, because rendering takes place while the user is typing, the render function must have reasonable behavior when the document ends with incomplete syntax like `{{tmpl|x=`.

Most of the Wiki implementation consists of the hand-written parser and renderer. They are written in a straightforward ML fashion: As a collection of mutually recursive functions using datatypes and pattern matching to represent tokens and parsed phrases.

The parser was originally a major performance bottleneck in the Wiki application—the original version was too slow to be used for anything more than a few dozen characters. This is due to the poor design and performance of JavaScript in general, and

Figure 6.4: A screenshot of the Wiki application, while editing a page. The editable text box at the bottom is the page's source code and the display at the top is its rendered version.

the quality of the JavaScript code produced by ML5/pgh. Since I don't have any control over JavaScript, improvements could only be made to the compiler. Many of the optimizations in ML5/pgh were included to specifically address problems with the code generated for this parser. The biggest improvement came from using a more eager scheduling strategy that executed several basic blocks in a thread before yielding (Section 5.5.3). This is because the tokenization code is a small tight loop over the input string, which otherwise causes an expensive yield for every character. Including a primitive `intcase` construct in the CPS language improved the inner loop, which was previously compiled as a series of `if` tests for character equality. Many more optimizations are possible for the compiler, but what is currently implemented is enough for the Wiki rendering code to run at a tolerable speed.

A consequence of a slow renderer is that we can get very bad behavior from race conditions and overeager updates. For example, if every keypress triggers a render (which can take up to a second for medium-sized pages), we will certainly have multiple renderers running concurrently on different input, each attempting to update the display with an outdated version of the page. The Wiki is therefore designed in a different way. We maintain two reference cells that store time values: the time that the page source was last modified, and the time that we read the page source that is currently rendered and displayed. Keypresses just update the modified time. If the modified time is more recent than the rendered time, then we must begin a new rendering thread to update the display. We perform this check at periodic intervals by using JavaScript timers. (These are just like the event handlers we place on UI elements, but that trigger after a specified delay.) Each check and potential render is guarded by a lock that prevents two outstanding renders from occurring simultaneously. This mutual exclusion routine

$$\texttt{maybe-with-lock}\ l\ f$$

checks to see if the lock l is currently held. If so, it discards f and does nothing. If it is not held, it atomically takes the lock and executes f, then releases the lock when f completes. The correctness of its implementation (which is part of a library) relies on knowing that certain sequences of operations do not yield; it would be much better to have language support for mutual exclusion and have the compiler guarantee the semantics.

We also have a reference cell that keeps track of the last time that the page was saved on the server, and a periodic timer that saves it. If the page has changed but has not been saved yet, then a message is displayed; the user can trigger a save or render manually by clicking one of the buttons in the interface.

The final interesting thing about the application is the treatment of templates. During the rendering of a page we might notice that we need the contents of another page in order to use it as a template. We could retrieve it from the server with `get`, but this would make rendering even slower. Instead, we keep a local cache of templates that we use for rendering. When rendering, if we do not have the value of a template yet, we render a placeholder. A separate thread retrieves the value of the template and updates the cache; whenever the cache is updated we also mark the page as modified so that it is

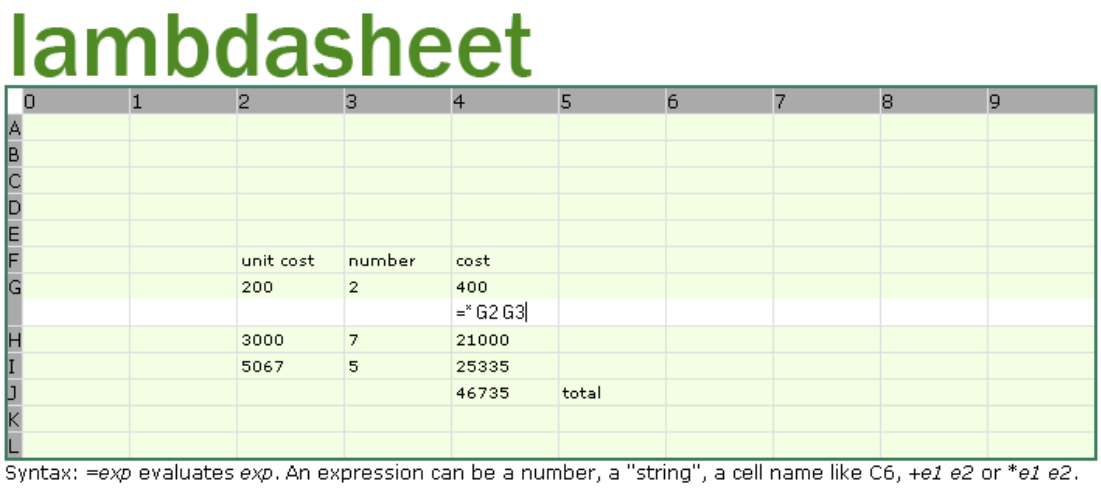

Figure 6.5: Lambdasheet screenshot showing the expression in cell G4 being edited.

rendered again. Finally, we install a hook for each template in the cache so that we can be asynchronously notified if one of them is modified (by another user, for example).

Aside from the mediocre performance of rendering, this wiki is fairly close to what a small real wiki would offer. Perhaps the only major missing feature is a way of tracking the history of each page—which would be easy to implement if the server had support in its database interface. On the negative side, the locking scheme used by the implementation is more complicated than I would like for such a small application. Writing concurrent programs is difficult, but high level structures for correctly implementing common idioms (at a minimum) would be desirable here.

6.4 Spreadsheet

The next application is a simple spreadsheet called Lambdasheet. A screenshot appears in Figure 6.5. It is a 12×12 matrix (the screenshot is truncated for space) containing named cells. Each cell contains an expression which may include the values of other cells. An expression can be a literal number or string, a cell name like `G4`, or an arithmetic expression in prefix form. For example, as shown in the screenshot, the expression in cell `G4` is `* G2 G3`, the unit cost times the number of items. Each cell also has a value derived from its expression. By default the spreadsheet is displaying only the values. Clicking on a cell allows the user to edit the expression. (Lambdasheet was co-written with Rob Simmons.)

The expressions in lambdasheet are stored in the database on the server, and as the values of HTML input boxes inside the DOM. These input boxes are hidden except when the user clicks to edit a cell. We synchronize the values of these cells with the database using a hook for each cell. (This means several users in different instances can edit the spreadsheet at the same time.) In addition to the expressions, we have an array of the same size as the matrix, which stores the last evaluated value of each cell's expres-

sion. We don't use the DOM to store the authoritative version of the values, because we want to distinguish the string value 3 from the numeric value 3 (even though they both display the same), for example. A loop runs periodically over all of the expressions in the spreadsheet, evaluating them using the current values of the cells in the value array, and updating those. This allows expressions to be recursive, with each successive pass computing the next iteration. Expressions are parsed and represented as ML datatypes, and evaluation is a straightforward pattern-matching ML function.

Lambdasheet is interesting in that it stresses the network performance of ML5. For example, if we used polling to update each cell from the server rather than asynchronous notification, then we would poll 144 times in each period! Asynchronous notification makes this efficient and reduces the update latency. Another instance of this is as follows: In each of these applications, at the point that we register a hook for a key, we also want to get the original value of that key. The typical pattern for this is to execute the hook function (as if the key had changed) at the same time we attach it. An early version of Lambdasheet did exactly this, but this resulted in very bad performance when the spreadsheet was loaded, as 144 round-trips from server to client to call the `updated-cell` function were initiated. Fortunately, this was easy to solve. As we register the hooks on the server, we insert any non-empty keys into a list of key-value pairs. Then, we return only this list to the client in a single message—its type is mobile, as a list of pairs of mobile types. On the client we map the `updated-cell` function over the list, registering the initial values all at once. The code is only a few lines long, and completely solves the performance issue by using data structures and higher-order functions.

6.5 Summary

In this chapter I presented a few web applications built with ML5. They all share the common feature of connecting local resources (a user interface in the web browser and the server-side database), making them true distributed computations. In some applications we also perform substantial computation on the client. Although the applications are small, they resemble real web applications in everyday use, with the major difference being features and scale. On the other hand, our high level language makes it simple to do things (such as threading and asynchronous notifications) that normally take a large engineering effort using conventional tools.

A natural question is, how did ML5 help us in building these? It is difficult to evaluate this question objectively, because to some extent language choice is a matter of taste, skill level varies greatly between programmers, and measures of code complexity or programmer effort (such as lines of source code) are very coarse. However, I will nonetheless argue for ML5's benefits. As mentioned, its high-level language features like `get`, higher-order functions, first-class continuations and algebraic datatypes with pattern matching make programs shorter and simpler to express. Second, even when used by experts (such as the language designer!), the type system does exclude programs with erroneous use of local resources in practice. A handful of times while

developing each of the applications, I encountered type errors (sometimes initially believing them to be bugs in the compiler) that indicated unsafe programs. Since no type system can be exact, the rate of false positives is also important. For these examples, I never found the type system to prevent a safe program that I wanted to write. Finally, and most importantly, ML5's modal type system yields a logical way of organizing a program according to the places involved (worlds), their local resources (modal code and data), and the computations and values that they share (valid code and data). This simply leads to better programs.

There are still some problems with ML5 and its implementation that make it inappropriate for building production-quality applications today. We have discussed some already, such as performance and language support for concurrency. In the next chapter—which concludes the dissertation—I discuss concrete future improvements as well as more speculative ideas. I also discuss related work, including some that could serve as a way of addressing some of these deficiencies, and work that could potentially benefit from the ideas prevented so far.

Chapter 7

Conclusion

In this chapter I conclude with a comparison of ML5 to related work (Section 7.1) as well as a discussion of some outstanding research problems and ideas for the future (Section 7.2).

7.1 Related work

Distributed computing is a large area of research and practice with differing ideas of what constitute it. Therefore, I will not attempt to compare ML5 to every language and tool. Instead, I will concentrate on comparing specific related work when it shares at least one of three salient features with this dissertation: Being based on modal logic, being a distributed ML-like language, or being a unified language for web applications. (Some related work is also discussed in the context of future work, in Section 7.2.) This leaves out many apparently related projects, so I will first situate ML5 in general relation to these as follows.

Located programming. ML5's main contribution to the design of languages for distributed computing is a logical, type-based account of *located programming*: Programming in a uniform language with an explicit notion of place (Chapter 2). Many programming languages used for distributed computing identify hosts in the computation using URLs (represented by strings) or IP addresses. These are reasonable run-time representations, but seldom do they find their way into the type system. This situation is not well-suited for located programming, because the programmer is unable to express the differing perspectives on code and data and his program except by informal means like naming conventions and documentation. An expressive type system is useful for structuring one's program in a logical way, but programmer discipline is not the only thing at stake. Lack of support for locations in the type system also leads to problems with the design of the language's distributed features. A common symptom of such inexpressiveness is the premature termination of a program that attempts to send a reference to one of its own local resources to another party in the computation. To be concrete, suppose that we have two parties in the program, A and B. A has a local open

file descriptor f that it wants to send to B. (This situation is not artificial; it is particularly common when f is within the environment of a closure.) If it tries, the program is either excluded statically (rare, except when the set of types of data that are allowed to be sent is extremely limited) or fails at runtime during marshaling. Both behaviors are needlessly conservative, because this program is safe as long as B does not try to read from or write to A's file descriptor f. It can safely send f back to A, for example.

Let us diagnose how a missing language concept of location leads to this result. Without being able to mention the principals A and B, "A's file descriptor" and "B's file descriptor" just become "file descriptor." (These English phrases correspond to the typing judgments of the programming language, because we use the type system to classify code and data.) Because the language must prevent an access to f from B's code, it must make sure that a file descriptor always refers to a local file. The standard way to do this is to identify the fault as being at the moment that f is sent to B. It is at this moment that we change perspective (running on A to running on B) and therefore that our notion of (local) "file descriptor" changes as well. Therefore f is no longer a file descriptor—the program must be rejected (statically or dynamically).

Located programming allows us to think more clearly about this situation, because it allows us to express the multiple simultaneous perspectives. Not only can we be explicit about where the code is running (by using the modal typing judgment on code), but we can simultaneously manipulate data that belong to various principals (by using the typing judgment on hypotheses, and using the types it engenders to describe data). In particular, code running on A can safely hand off a file descriptor (f:`file@`A) to code running on B. It does this by first encapsulating it to mark that it belongs to A (`hold` f:`file at` A`@`A), then observing that the marked datum is portable (`file at` A `mobile`) and thus also makes sense from B's perspective (`file at` A`@`B). Although B can now manipulate the encapsulated value (perhaps sending it back to A), it is statically prevented from erroneously reading from it, as desired. This mindset allows us to decouple the implementation technique of marshaling (which now never fails) from the semantic quality of mobility.

Some languages address this problem in another way: transparent mobility. In such a language the above program does not fail, but if B reads from f, then the system automatically forwards these reads to A and the results back to B. In doing so, the type "file descriptor" now means "global file descriptor"—forcing a universal viewpoint of data. This is legitimate and often what the programmer wants, but is not *only* what a programmer wants. In ML5, we allow the programmer to implement global resources by using local resources and mobility. These global resources can be given this universal viewpoint through the validity judgment. Therefore, such global resources are still expressible when using located programming (perhaps provided as libraries), but we additionally have the ability to be specific when we desire.

Process calculi. A long line of research in distributed programming languages focuses on adding distribution to process calculi (such as the π-calculus [74, 76]). This is usually accomplished by adding explicit locations to the language. For example, the SAFEDPI

calculus [56] extends the π-calculus to support distribution by wrapping processes with explicit locations. A process can migrate to a new location via a construct `goto`, and new locations can be created at runtime. The language addresses the control of access to local resources (represented as names) by permitting a location to restrict the types of code that can migrate to it. The type system is sophisticated, allowing the specification of complex behaviors.

Nomadic Pict [125, 135] extends the π-calculus based programming language Pict [109] to distributed processes. Each process again is marked with an explicit location. The low-level calculus has only local communication primitives. The high-level language implements unrestricted communication between processes at different locations by translation to the low-level language through various strategies.

This line of research on process calculi attempt to address similar problems as does ML5: locality; mobility; local and global resources. They also begin from foundational calculi and their semantics rather than implementation concerns. However, the feel of the work is very different, making a direct comparison difficult. In general, the primitives of *process* and *channel* lead to a natural focus on concurrency and communication—two facets that we have deemphasized in this work. Lambda 5's basis in logic instead naturally leads to a focus on types, data, and reduction-based operational semantics. It also permits straightforward integration with functional programming languages like ML that are already based on logic, and is compatible with traditional compiler techniques. Because my goal is to give an account of the *spatial* component of distributed computing—and to implement it—the logical approach fits well. Nonetheless, we have observed that concurrency support is important, even in the simple applications I have experimented with. Perhaps process calculi can provide the foundation for concurrency support in ML5?

7.1.1 Modal logic in distributed computing

Other lines of research have investigated the use of modal logic for distributed computing. One of the earliest is Borghuis and Feijs's Modal Type System for Networks [10]. It is a logic and operational semantics—by way of compilation into shell scripts!—for network tasks with stationary services and mobile data. They use $\Box$, annotated with a location, to represent services. For example, $\Box^o(A \supset B)$ means a function from A to B at the location (world) o. This corresponds to the type $(A \supset B)$ `at` o in Lambda 5. However, with no way of internalizing mobility as a proposition, the calculus limits mobile data to base types. Services are similarly restricted to depth-one arrow types. In contrast, ML5 permits higher-order distributed programming in its full generality.

Cardelli and Gordon [15] provide an example of using modal logic for reasoning about programs spatially, later refined by Caires and Cardelli [13, 14]. They do not take a propositions-as-types view of their logic, instead using it as a way of stating and proving properties about distributed programs. Their object language is a process calculus, mobile ambients [15]. Therefore, their classical modal logic is very different from the one that we use, including connectives for stating temporal properties, security properties, and properties of parallel compositions. In contrast, ML5 is based on interpreting

the propositions of modal logic as types, and giving them a computational interpretation based on proof reduction. This gives us a type theory (which allows us to logically structure programs and prove certain properties about the language, like type safety) for our programming language, rather than a theory for proving properties of specific programs.

Moody [80, 81] gives a lambda calculus based on the constructive modal logic S4 of Pfenning and Davies [107]. This logic's accessibility relation is reflexive and transitive, but not symmetric, and the presentation relies on judgments A true (here), A valid and A poss rather than the indexed truth judgment of IS5$^{\cup}$. As a result, worlds do not appear explicitly in the judgments or types. The propositions are therefore somewhat "lossy" in the sense we discussed in Section 4.1. As a result, the $\Box$ and $\Diamond$ connectives are interpreted as representing *potential* mobility and locality, with a type-safe process-style operational semantics to match.

More closely related is Jia and Walker's computational interpretation of hybrid S5 [62]. As we have discussed, this was the source of our `at` modality. They also have a notion of validity, where the $\Box$ connective is the source of valid hypotheses. (Unlike Lambda 5's validity judgment and $\mathfrak{Z}$ modality, these do not bind a world variable.) The biggest difference in Lambda 5 is the presence of the `get` structural rule (and the mobile judgment), and its restriction of the other rules to act only on local data. Without these, Jia and Walker's calculus has a process-style operational semantics that requires synchronization across worlds. In contrast, Lambda 5's operational model can be given in terms of proof reduction, with standard substitution of values for variables being the main force of computation.

Because in Lambda 5 and ML5 evaluation can take place in different locations, we are not always in a position to evaluate an expression into its value. Therefore, we found it important to distinguish between the modal typing judgment as applied to expressions (code) and values (data). For example, the $\mathfrak{Z}A$ type classifies mobile *values* of type A, and the typing rules for the expression `hold` M and value **held** v are different. Park [100, 101] studied the problem of mobile values in an S4-style modal logic by introducing a new modality. His type $\boxdot A$ classifies computations that return mobile values of type A. (In contrast, $\Box A$ classifies mobile computations that return possibly non-mobile values of type A). This modality was the original inspiration of Lambda 5's $\mathfrak{Z}$ modality. The $\mathfrak{Z}$ modality is somewhat more simple, in that we do not have a validity judgment for expressions—$\mathfrak{Z}A$ represents not a computation producing a mobile value of type A, but an already evaluated mobile value of type A. (Since ultimately in the compiler we require "values" to include some forms of pure computation, this is perhaps not a difference after all.)

Park refines this calculus to $\lambda^{\mathsf{PC}}_{\boxdot}$, a call-by value parallel language with communication primitives [102]. By using the validity judgment and its internalization as a type, the language is able to statically prevent local resources (references) from being sent through a channel to another thread. They do this by splitting a communication channel into two distinct parts: its read half and its write half. The read half remains fixed to the thread that created it, and the write half is allowed to be a global value. Channels are restricted such that only global values can be sent over them. This is overconserva-

tive in the sense described above: Programs are rejected because they try to *share* remote resources, not because they try to improperly *use* them. Specifically, if the read half of a channel always belongs to (say) thread g_1, any value that is sent along the write half always ends up in g_1. Therefore, the channel ought to be restricted not to carry a global value, but a value that makes sense specifically in the thread g_1. There is no way to express the condition more tightly in $\lambda^{\mathsf{PC}}_{\square}$, since when typechecking the code for the thread g_2, we have only two typing notions available to us: "here" (wrong, since here means g_2, not g_1) and "global."

7.1.2 Distributed ML-like languages

The ML family of programming languages has been extended in multiple ways to allow for distribution. Distributed ML [22, 64] concentrated on the concurrency and failure aspects of distributed computation, with abstractions such as multicast port groups and fault-tolerant port colonies. I deemphasize these in the design of ML5. Early work by Ohori and Kato on dML showed how to provide marshaling and unmarshaling functions in the presence of polymorphism, by a technique similar to type passing [97]. The dML language only allowed marshaling of base types, creating proxies at run-time for values of higher type. Harper and Morrisett remarked on how to implement this marshaling strategy using a more general type passing technique [53].

Facile [132, 133] is a concurrent distributed programming language based on Standard ML and the CCS [73] process calculus. Facile uses SML's type system almost directly; its features for distribution are expressed in terms of a module with an abstract type `nodeid`, along with routines for manipulating nodes and launching processes on them. The Facile model has each host running a number of processes, which communicate with each other over typed channels. These typed channels (similar to those in CML [115]) are not restricted to simple data; they can transmit functions, channels, and abstract types. There is no particular facility for safely delineating local resources as in ML5: Reference cells are copied when they are marshaled, and other kinds of local references can cause runtime failures if used improperly.

D'Caml [142] is a distributed extension of the O'Caml dialect of ML [96] that presents a different model of computation: a single parallel computer with distributed shared memory. In such a model, the ML type system suffices to ensure the proper use of local resources, because it becomes the compiler and runtime's job to make sure that every use succeeds. ML5 is lower-level, in that the distribution and communication are explicit.

JoCaml [63] is an extension of O'Caml with primitives for concurrent, distributed programming inspired by the Join Calculus [40]. It allows the creation of hierarchically-structured "locations," which contain bindings and processes that are permanently localized to that location. However, locations themselves are mobile; they can move from one parent location to another using the `go` statement. Since locations can move, they are perhaps better thought of as delineating a region of the running program than corresponding to the physical sites of computation. In comparison, ML5's locations—worlds—are unstructured names for fixed places; we achieve mobility by demonstrat-

ing in the type system that a piece of code or data is well-formed independent of its location.

Acute [123, 124] for distributed computing based on O'Caml. Rather than commit to a high-level model of interaction like ML5's `get`, they provide type-safe access to low-level features like marshaling so that the programmer can implement his own. Acute's goal is somewhat different, in that it allows for the development of programs without full static knowledge of the environment in which they will run. This is important for distributed applications where hosts may be discovered at runtime or that are upgraded in a piecemeal fashion. To accomplish this, Acute supports two ways of resolving a resource in a piece of code that arrives at a remote host. A representation of the resource can simply be sent along with the code, or the resource can be rebound to a local one—appropriate for library code, for example. Acute uses hashes of module definitions and other techniques to guarantee type safe marshaling in the presence of abstract types. This technique was later extended to account for type generativity and polymorphism in HashCaml [7, 8]. ML5 does not attempt to maintain abstraction at runtime, instead assuming that all of the code was produced by the same version of the source program. Like Acute and HashCaml, the ML5 intermediate languages use type representation passing to implement type-safe unmarshaling. However, Acute and HashCaml are built on the O'Caml marshaling routine, which already has enough information without type representations to guide the marshaling process. Therefore, type representations are only used for a compositional run-time equality (rather than a recursive inspection of their structure), so they are represented much more compactly as 256-bit hashes. In ML5 we are using the type representations as disembodied tags for the native representations, which gives us more flexibility but also makes them more costly.

The Alice language [117, 118] is an extension of Standard ML to permit "open programming"—dynamic integration of software components in a distributed program. Its design is therefore closely related to Acute's, allowing resource dependencies to be sent along with components or dynamically rebound upon arrival at a site. Alice's facility of pickling [119] allows for the type-safe marshaling of arbitrary values (including higher-order code and modules), but fails dynamically if it attempts to marshal a local resource.

ML5 is based on a more static view of distributed programs. It is clear that this suffices for at least simple web applications, since the set of worlds and the resources involved is small and fixed. However, because they are based in type theory, support for dynamic components such as in Acute and Alice would probably integrate into ML5 in a clean way. In fact, Alice-style higher-order modules might give us a natural way to discover worlds at runtime, by allowing abstract worlds to be a third component of modules along with values and abstract types.

7.1.3 Languages for web applications

With the rise of the web as an application platform, there has been much interest in it from programming language designers. One of the major focuses has been on creating

a unified programming language for web applications. Current development practice often involves several programming languages and tools within the same application (JavaScript, Java, PHP, SQL, etc.) and ad hoc structures for communicating between them (hand-written text- or XML-based protocols, string-based SQL query generators, etc.).

QHTML [33] is a module for the Oz language [127] that allows it to be used for web application development. It has a language for declarative GUI specifications (from which HTML is derived) and a thread model for responding to UI events. Thread mobility between client and server is transparent. QHTML works by embedding a Java applet within the web page and using it to run Oz code and as the liaison between the client and server.

Links [21] is a language designed particularly for web programming. It has a JavaScript-like syntax but an ML-like semantics. Links has explicit distribution: Functions may be annotated as *client* or *server*, and Links allows calls between client and server code. However, its type system does no more to indicate what code and data can be safely mobile, and marshaling can fail at runtime. On the other hand, Links has many features (such as a facility for embedding and checking XML documents and generating database queries from Links code) that make typeful web programming easier. It additionally supports a mode of page-oriented application where all of the session state is stored on the client, as marshaled data inside of hyperlinks and forms. This can be used to increase performance by reducing the amount of persistent storage on the server. In contrast, ML5 only supports "AJAX" style web applications (*i.e.*, a single page that the user never leaves), because our symmetric computational model requires that the server be able to contact the client at any time.

Hop [122] is another unified web programming language, based on Scheme. Hop has constructs for embedding a client side expression within a server expression and vice-versa, analogous to `get` in ML5 (but specific to the two-world case). The chief difference is simply that Hop is untyped, and thus subject to run-time failures.

Swift [19] is a web programming language based on the Java variant Jif [93]. In Swift, a web program is annotated with information flow properties about the secrecy and authenticity of data. The program is then compiled (if possible) into client-side JavaScript and server-side Java that respect the security annotations given. Distribution and communication are not explicit in the program, so compilation is achieved by program partitioning. The program is partitioned in such a way as to minimize communication. Swift represents one solution to an important security problem that faces ML5 and other distributed programming languages; this is further discussed in Section 7.2.

7.2 Future work

There is much potential for future work on modal type systems and ML5.

7.2.1 Modal type systems

Interpreting the modality. Lambda 5 was designed as a computational modal logic where worlds are interpreted as places in a distributed computation. However, at this abstract level there is nothing that forces us to interpret the worlds spatially. Indeed, there have been many other uses of computational modal logics, interpreting the worlds as security principals [1, 2, 44, 65], stages of computation in a metaprogramming language [29], steps in a time-ordered sequence of computations [28, 35, 57], possibilities in the decision making processes of agents [114], etc. Broadly speaking, whenever a setting has multiple simultaneous perspectives, a modal logic may be good way to reason about it and a modal lambda calculus may be the basis of a natural programming language. Lambda 5's decomposition into locally-acting introduction and elimination rules and the perspective shifting `get` structural rule may be useful for these other interpretations of modal logic as well. Of course, we have to make sure that our choice of accessibility, as well as semantic notions such as the mobile judgment, are appropriate for the application. For example, a temporal logic in which time may symmetrically step backwards and forwards would be suspicious!

Wide-area physical distribution is also not the only spatial interpretation of worlds. Computer architectures are becoming increasingly non-uniform, leading to a smaller-scale notion of locality. Even though virtual memory machines present a flat address space abstraction, the performance characteristics differ substantially between data that reside in register-shadowed stack slots, L1 and L2 caches, main memory, memory-mapped files, and disk. (The semantics of these can differ as well, for instance with floating-point precision and memory coherence in multiprocessor environments.) A type system for a low-level language could allow the programmer to make these locations explicit using the spatial modality of Lambda 5. The Java-based X10 language [18] is designed for such non-uniform memories, but its notion of place is not logically derived.

Accessibility. In our distributed interpretation of Lambda 5 the accessibility relation is realized as the connectivity between hosts. We chose a universal accessibility relation (every host can access every host) for its simplicity and because many networks are universal in this sense. However, because of technologies like NAT [128] and security policies like the JavaScript same origin policy [120], many real networks are not symmetric or transitive. (It is somewhat ironic that the prototype ML5 implementation is for the web, one of the most restrictive networks in wide use!) To use ML5 on such networks, we currently build support for universal accessibility into the runtime system. Another approach would be to use an accessibility relation with fewer structural properties, so that accessibility resembles the low-level connectivity that we actually have available. (The overlay network could then be written as a program in the language, rather than in the runtime system.) Assumptions of the existence of worlds would be augmented with accessibility assumptions, and the notion of global "addresses" (the dynamic permission to access a world) would be replaced with local "routes" (a structured path from one world to another). Doing this for ML5 would be somewhat tricky,

because the completeness of the logic with local introduction and elimination rules depends on the structural rule `get`. The `get` rule has some notion of symmetry built into it, because it transfers the locus of reasoning to the remote world and then returns a proposition. Therefore, its analogue in a non-symmetric network is not clear. We can see this in the translation of `get` to the CPS language `go`, in which we bind the return address as a valid variable. Interestingly, the `go` construct therefore does not have such symmetry built in, so the CPS-style language is more straightforwardly compatible with a non-universal accessibility relation.

7.2.2 ML5 and its implementation

For the ML5 language, there are numerous improvements that could be made. I have remarked on some of the obvious missing features, such as modules and high-level concurrency support. I believe a Standard ML style module system [31, 55] could be added in a fairly orthogonal way. Acute and Alice provide a roadmap for integrating modules and distribution. More interesting would be the presence of abstract worlds within modules; these would give us a natural way to provide compositional distributed services as libraries. It is less obvious what to do for concurrency; a variety of different mechanisms have been studied throughout the history of computer science, from process calculi to software transactional memory.

Multiple worlds. Our type theory naturally supports an arbitrary number of worlds, and most of the compiler does, as well. Adding the ability for a program to access many different servers would just be a matter of adding runtime support for it. This would take the form of an overlay network to circumvent JavaScript's same-origin security policy. Applications that have a single client and multiple servers are not as easy to motivate as applications with a single server and multiple clients; we have already seen some examples that would benefit from this latter model. This would take a bit more design work, because we currently consider the thread of control to begin on the (one) client. To support multiple clients, we would want to begin the thread of control on one server, and then instantiate a piece of world-polymorphic code for each new client that connected. This would mean adding a primitive for handling a new HTTP connection, so that some part of what Server 5 does would become part of the application code.

Fault tolerance. I have so far ignored the issue of fault tolerance, which is a major focus of research in distributed computing. For ConCert we built a system that automatically tolerated host failure by repeating a computation on another host [85]. We were able to do this because all computations were total, portable between machines, and automatically allocated by the runtime. The ML5 programming model is much lower-level in that the programmer explicitly chooses where a computation will run. In the case that the host is unreachable or fails while the computation is active, there is not much hope for automatic recovery since the computation would need to be run in that same failed place. A simple notification (by raising an exception, for example) could

inform the programmer and allow him to handle the failure in an appropriate way—perhaps explicitly choosing to re-run the computation on a different host. It is helpful that interaction between hosts only happens via the `get` construct, so the programmer can identify the potential failure points in his program and know what actions are possible once a failure has been detected. Failure tolerance is very important for large-scale computations, because the probability of failure rises as the number of hosts involved increases. However, for our example domain of web applications it is not a serious problem: If one of the two hosts fails, then there is not much to do except abort the application. (The ML5/pgh runtime currently does detect communication failure and display an error message.)

Garbage. Distributed garbage collection [111] is a notoriously difficult problem, and I have avoided it completely in ML5. Therefore, although the garbage collectors in the JavaScript and Server 5 runtimes can reclaim local garbage, once a reference to a resource escapes a world it can never be freed. This manifests itself as a continually-growing table of desiccated pointers (Section 5.5.4). Fortunately, web applications tend to be short-lived, so this has not been a limiting factor for the example applications. For any long-running computation, we would need a distributed garbage collector. Nothing about the design of ML5 makes this problem any more difficult than usual.

Marshaling. The ML5/pgh runtime does only simple data validation during unmarshaling. Therefore, if the marshaled data is accidentally or maliciously corrupted, this can lead to runtime failures. Because we unmarshal with respect to a type representation, we can easily add more checks to ensure that the runtime failure happens at unmarshaling time rather than later. Currently the runtime assumes that the network is reliable and the participants trustworthy. (This is mainly because I do not address more difficult security issues, which are discussed below.) There are a variety of ways that we could improve the robustness of unmarshaling, for example by using cryptographic signatures to maintain the authenticity of data [3].

Certification. Related to the previous point is code certification. It would be possible to remove the need for all of the code to be present before the program ran, by allowing code to be marshaled along with a proof of its type-correctness, using proof carrying code [95] or typed assembly language [23, 25, 82]. If we were using certified code, we could do away with JavaScript sandboxing and run native code (via a browser plugin), which could dramatically improve performance and reduce the size of the trusted computing base. The fact that the ML5/pgh compiler is type-directed is an important precondition for certification.

Security. A more serious security problem faces us when some of the participants in a distributed computation are malicious. This is a realistic scenario for web applications—although the application author can usually assume that the server is secure, web sites are often deployed publicly. JavaScript code intended to run on the client is actually

under the complete control of an attacker. He can inspect its source and cause it to behave arbitrarily, and invoke any continuation on the server for which he is able to craft acceptable arguments. This is true of any distributed system where some hosts are controlled by attackers, and the programmer must defend against this by not trusting (and explicitly checking) data and code it receives from the client. For example, take the following program:

```
extern val format_hard_drive : unit -> unit @ server
extern val prompt_password : unit -> string @ home

do if prompt_password () seq [secret]
   then let in
          alert [Formatting...];
          from server get format_hard_drive ()
        end
   else alert [Wrong password!]
```

This program runs at home, prompting the user for a password (`seq` is the string equality function). If it matches, then it travels to the server and formats its hard drive. This program is clearly insecure: Because the client does not have to run the code we give it, it can force the string equality to succeed (or inspect the source code or memory to learn the password) and therefore bypass the check. It is easy to diagnose this problem as an instance of improperly trusting the client to run the code we give it. For example, we could imagine giving the client the ability to modify any source code and read any data that is typed **@home** and then observe that it can replace the password check with `true`. Unfortunately, the attacker's capability for mischief goes beyond what we can explain at the source level, because the process of compilation from the high-level language is not fully abstract. For example, suppose we rewrite the above program to the following equivalent:

```
extern val format_hard_drive : unit -> unit @ server
extern val prompt_password : unit -> string @ home

val pass : string @ server = "secret"
val errormsg : string @ server = "Wrong password!"

put p = prompt_password ()

do from server
   get if p seq pass
       then let in
              from client get alert "Formatting...";
              format_hard_drive ()
            end
   else let put u = errormsg
        in from client get alert errormsg
```

```
end
```

In this version we have explicitly typed the variable `pass` at the server so that the client will not have access to it. We could arrange for the ML5/pgh compiler to encrypt data typed at the server, since we know it won't be used in other places. We then perform the password check on the server, where we trust the program to behave as written. Unfortunately, this program also has security problems. First is that the process of CPS and closure conversion leaves the client with a continuation and environment containing the values of the variables `pass` and `errormsg`. They may be encrypted, preventing the client from reading them directly, but he can modify the closure to swap the values of the two variables! When control returns to the server, it will either succeed with the password equivalence check (if the client typed in the string `"Wrong password!"`, which is now the value of `pass`) or will return to the client to display the `errormsg`—which is now the secret password. Alternatively, the client can invoke a different continuation directly (by guessing the code label and environment for it); for example returning from the `get` that displays the `Formatting...` message after the password check has been successful. Neither of these behaviors is easy to explain without considering the way that the program is compiled; the first modifies the contents of immutable variables, and the second involves a non-local flow of control.

Each of these problems is solvable by some means, using cryptography or other countermeasures to prevent the attack. However, a series of responses to anticipated attacks is not good enough: To build secure software, a programmer must be able to understand the range of behaviors that may occur in the presence of an attacker. I believe a solution to this problem would take the form of an "attack semantics" provided by the language and implemented by the compiler through a series of countermeasures. The semantics would describe at the language level the set of possible behaviors, so that the programmer can ensure that these behaviors do not include security breaches on the server. (The client will always be able to format his own hard drive, if he desires.) One example of such a semantics is Swift's information flow annotations: No matter how the client behaves, the language guarantees that he cannot (for example) read or influence the values of secret variables. Information flow is clearly appropriate for enforcing the secrecy of the password in the above example. For the sake of protecting the `format_hard_drive` routine, we might desire a more direct means of stating our policy, such as dynamically insisting on a proof that the password check had been performed successfully on the server in the past. (Swift accomplishes this by its *checked endorsement* construct.) In any case, such properties are inherently in terms of the principals (places) involved in the computation, and therefore I believe that our type system and semantics is an important ingredient for being able express and prove properties of programs in the presence of an attacker, and to develop mechanisms for building secure programs.

7.2.3 Web programming

ML5 specifically as a web programming language could be improved in a few ways, as we observed in Chapter 6. Two of the most common tasks in a web application are interacting with server-side resources (usually a database) and manipulating user interfaces on the client side. The current prototype has very rudimentary support for each of these.

Databases. Accessing a database is simple: We can create a datatype for SQL queries (for example) and use the network signature to expose an interface to a database engine of our choice. Since database access is so pervasive, we might consider language extensions. Links has support for database queries based on a technique from Kleisli [12, 144]: Semantically, a primitive database operation returns a stream of all results from the database, and standard features from the programming language are used to implement the query. The compiler translates the functional manipulation of the stream into the database language (such as SQL) to produce an efficient query. This has the advantage of being semantically lightweight, requiring almost no new language features. I do not like it, however, because it requires the programmer to anticipate when the compiler will be able to produce an efficient database query. Being able to reason about the efficiency of queries is critical when data sets are large.[1] Therefore, I prefer that the language have direct support for expressing the kinds of operations that the database can perform efficiently. The main benefit of a language extension in this case is that we can type-check the queries and their results. Such an embedding is fairly straightforward.

Client pages. The current way of creating and modifying the client user interface is dissatisfying. We use strings to write HTML concrete syntax directly, which is compact but not subject to any static checks. JavaScript event handlers are similarly represented as strings and are essentially untyped. We could build a high-level client library for interacting with the DOM using the features we already have, or consider language extensions that allow us to express XML documents using a convenient syntax and appropriate type system. There has been much interest in this problem from the programming languages community. JWIG [20] is a Java extension that uses a flow analysis to check that generated HTML is well-formed. Elsman and Larsen give a Standard ML library for constructing XHTML documents, which uses phantom types to ensure that the documents are well-formed [34]. WASH/CGI [131] is a monadic library for Haskell [103] for the creation of well-formed XHTML documents. It uses type classes to statically encode (most of) the type structure of XHTML. WUI [48] is a similar typed monadic combinator library for the functional logic language Curry [26]. The main purpose of WASH/CGI and WUI are to provide a language for sessions with the web server that are translated to uses of HTML forms and CGI. These features are not important for ML5, which is

[1] Database systems also perform substantial query optimizations as well. However, these optimizations are often explicitly specified and tools such as SQL's `EXPLAIN` statement exist to diagnose performance implications.

committed to AJAX-style [45] interaction. Hop [122] simply uses Scheme functions for each of the HTML tags to ensure that they are properly balanced; it is untyped. Links's syntax and type system are designed for the purpose of producing well-formed XML documents [21]. Gardner et al. [43] give a minimal formalization of DOM and a Hoare-like logic for stating properties and proving them of simple programs that manipulate the DOM.

With a complete library or language for writing XML DOM documents, we could replace the string-generating `say` with a construct that produces an abstract type of event handler, which can only be used in the appropriate contexts.

7.2.4 Conclusion

In this dissertation I have argued that modal type systems provide an elegant and practical means for controlling local resources in spatially distributed computer programs. The project is a complete study in programming language design, beginning from the identification of the problem (an inexpressiveness resulting a lack of multiple simultaneous viewpoints), to the design and implementation of a new programming language to address the problem, and the crafting of applications in that language to study its effectiveness.

The specific contributions are summarized as follows: I developed a spatial formulation of modal logic and derived the lambda calculus Lambda 5 from it. The logic admits cut and the calculus is given a type-safe distributed operational semantics based on standard substitution-based proof reduction. I then showed how this calculus could be extended to account for distributed control flow and global resources in a logical way. Each of these systems were formalized in LF and their metatheoretic properties proved in Twelf; the proofs are therefore machine-checkable. Next, I developed an abstract typed compilation technique for the modal lambda calculus, including CPS conversion and closure conversion. These transformations were proved to be sound, also in machine-checkable form. To argue for the practicality of modal type systems, I then designed a high-level language ML5 that combined features from ML with the new modal constructs. The new features integrate naturally into ML-style languages, retaining for example full polymorphic type inference. Using the abstract compilation technique I formalized, I then implemented this language as a type-directed compiler. The compiler is specialized to web applications, a particular kind of distributed computation involving two hosts: the web server and the web browser. The compiler produces code in different languages for the server and for the web browser, and a runtime system including a web server and type-directed marshaler ties them together. Using this implementation, I then developed a collection of applications to exercise the language's features and evaluate its effectiveness at solving the stated problem. Although there are a few remaining issues to be addressed before ML5 can be used for production-quality code, the results are encouraging. The language is expressive enough to build realistic applications, performance is acceptable with much room left for improvement, and its type system excludes runtime failures while encouraging a logical structure of programs based on the places (worlds), local resources (modal code) and shared com-

putations (valid code) involved. This leads to simpler, more reliable, and more elegant distributed programs.

Appendix A

Twelf proofs

A.1 Equivalence of Lambda 5 natural deduction and sequent calculus

```
%% Soundness and completeness of Lambda 5
%% Relative to IS5U Sequent Calculus

%%% Based on proofs by Frank Pfenning, 2004
%%% Revised and extended by Tom Murphy VII, 2007

world : type.                           %name world W w.
prop : type.                            %name prop A x.

% Propositions

=> : prop -> prop -> prop.              %infix right 8 =>.
! : prop -> prop.                       %prefix 9 !.
? : prop -> prop.                       %prefix 9 ?.
at : prop -> world -> prop.             %infix none 6 at.
& : prop -> prop -> prop.               %infix left 7 &.
| : prop -> prop -> prop.               %infix left 7 |.

% Natural deduction

mobile : prop -> type.                  %name mobile M m.
@ : prop -> world -> type.              %name @ N.
%infix none 1 @.

!mob  : mobile (! A).
?mob  : mobile (? A).
atmob : mobile (A at W).
&mob  : mobile A -> mobile B -> mobile (A & B).
|mob  : mobile A -> mobile B -> mobile (A | B).


% Structural
get : mobile A -> A @ W -> A @ W'.

% Implication
=>I : (A @ W -> B @ W) -> (A => B @ W).
=>E : (A => B @ W) -> A @ W -> B @ W.

% Necessity
!I : ({o:world} A @ o) -> ! A @ W.
```

```
!E : ! A @ W -> A @ W.
!G : ! A @ W' -> ! A @ W = [a] get !mob a.

% Possibility
?I : A @ W -> ? A @ W.
?E : ? A @ W -> ({o:world} A @ o -> C @ W) -> C @ W.
?G : ? A @ W' -> ? A @ W = [a] get ?mob a.

% Hybrid
atI : A @ W -> A at W @ W.
atE : A at W' @ W -> (A @ W' -> C @ W) -> C @ W.
atG : A at W @ W' -> A at W @ W'' = [a] get atmob a.

% Conjunction
&I : A @ W -> B @ W -> A & B @ W.
&E1 : A & B @ W -> A @ W.
&E2 : A & B @ W -> B @ W.

% Disjunction
|I1 : A @ W -> A | B @ W.
|I2 : B @ W -> A | B @ W.
|E : A | B @ W -> (A @ W -> C @ W) -> (B @ W -> C @ W) -> C @ W.

% non-local disjunction elimination is a derived form
|Er : A | B @ W' ->
        (A @ W' -> C @ W) ->
        (B @ W' -> C @ W) -> C @ W
 = [ab'][a'c][b'c]
     |E (get (|mob atmob atmob)
           (|E ab' ([ha] |I1 (atI ha))
                   ([hb] |I2 (atI hb))))
        ([ha'] atE ha' a'c)
        ([hb'] atE hb' b'c).

% %%%%%%%%%%%%%%%%%%%%%
% IS5U sequent calculus
% %%%%%%%%%%%%%%%%%%%%%

hyp : prop -> world -> type.            %name hyp H h.
conc : prop -> world -> type.           %name conc D.


init : hyp A W -> conc A W.

=>R : (hyp A W -> conc B W)
       -> conc (A => B) W.
=>L : conc A W
       -> (hyp B W -> conc C U)
       -> (hyp (A => B) W -> conc C U).


!R : ({o:world} conc A o)
      -> conc (! A) W.
!L : (hyp A W' -> conc C U)
      -> (hyp (! A) W -> conc C U).


?R : conc A W'
      -> conc (? A) W.
?L : ({o:world} hyp A o -> conc C U)
      -> (hyp (? A) W -> conc C U).


atR : conc A W ->
       conc (A at W) W'.
atL : (hyp A W -> conc C W'') ->
```

```
       (hyp (A at W) W' -> conc C W'').

&R : conc A W -> conc B W ->
       conc (A & B) W.
&L : (hyp A W -> hyp B W -> conc C W') ->
       (hyp (A & B) W -> conc C W').

|R1 : conc A W -> conc (A | B) W.
|R2 : conc B W -> conc (A | B) W.
|L : (hyp A W -> conc C W'') ->
     (hyp B W -> conc C W'') ->
     (hyp (A | B) W -> conc C W'').

% for some reason Twelf's subordination relation is
% too conservative here. Obviously hyp can't appear
% in mobile...
stripmh : (hyp B W -> mobile A) -> mobile A -> type.
%mode stripmh +M -M'.

- : stripmh ([h] !mob) !mob.
- : stripmh ([h] ?mob) ?mob.
- : stripmh ([h] atmob) atmob.
- : stripmh ([h] &mob (M1 h) (M2 h)) (&mob M1' M2')
 <- stripmh M1 M1'
 <- stripmh M2 M2'.
- : stripmh ([h] |mob (M1 h) (M2 h)) (|mob M1' M2')
 <- stripmh M1 M1'
 <- stripmh M2 M2'.

%%% Admissibility of Cut

cut : {A:prop}
      conc A W ->
      (hyp A W -> conc C U) ->
      conc C U ->
      type.
%mode cut +A +D +E -F.

% Expansion
exp : mobile A -> hyp A W -> conc A W' -> type.   %name exp AM.
%mode +{A:prop} +{M:mobile A}
       +{W:world} +{W':world}
       +{H:hyp A W} -{C:conc A W'}
       exp M H C.

- : exp (&mob Ma Mb) H (&R (&L ([ha][hb] Na ha) H)
                           (&L ([ha][hb] Nb hb) H) : conc _ W)
 <- ({ha} exp Ma ha (Na ha))
 <- ({hb} exp Mb hb (Nb hb)).

- : exp !mob H (!R [w] !L ([ha] init ha) H).

- : exp ?mob H (?L ([w][ha] ?R (init ha)) H).

- : exp atmob H (atL ([ha] atR (init ha)) H).

- : exp (|mob Ma Mb) H (|L ([ha] |R1 (Da ha))
                           ([hb] |R2 (Db hb)) H)
 <- ({ha} exp Ma ha (Da ha))
 <- ({hb} exp Mb hb (Db hb)).


shift : mobile A -> conc A W -> conc A W' -> type.
%name shift OD.
%mode +{A:prop} +{M:mobile A}
       +{W:world} +{W':world}
```

```
       +{H:conc A W} -{C:conc A W'}
       shift M H C.

- : shift atmob (atR D) (atR D).
- : shift !mob (!R D) (!R D).
- : shift ?mob (?R D) (?R D).
- : shift M (=>L D1 D2 H) (=>L D1 D2' H)
 <- ({hb} shift M (D2 hb) (D2' hb)).
- : shift M (!L D H) (!L D' H)
 <- ({ha} shift M (D ha) (D' ha)).

- : shift M (init H) D
 <- exp M H D.

- : shift (|mob Ma Mb) (|R1 D) (|R1 D')
 <- shift Ma D D'.

- : shift (|mob Ma Mb) (|R2 D) (|R2 D')
 <- shift Mb D D'.

- : shift (&mob Ma Mb) (&R Da Db) (&R Da' Db')
 <- shift Ma Da Da'
 <- shift Mb Db Db'.

- : shift M (|L D1 D2 H) (|L D1' D2' H)
 <- ({ha} shift M (D1 ha) (D1' ha))
 <- ({hb} shift M (D2 hb) (D2' hb)).

- : shift M (&L D H) (&L D' H)
 <- ({ha}{hb} shift M (D ha hb) (D' ha hb)).

- : shift M (?L D H) (?L D' H)
 <- ({w}{ha} shift M (D w ha) (D' w ha)).

- : shift M (atL D H) (atL D' H)
 <- ({ha} shift M (D ha) (D' ha)).

% Initial cuts
ci_l : cut A (init H) ([h] E h) (E H).
ci_r : cut A D ([h] init h) D.

% Principal cuts
c_=> : cut (A1 => A2) (=>R ([h1] D2 h1))
           ([h] =>L (E1 h) ([h2] E2 h h2) h) F
        <- cut (A1 => A2) (=>R ([h1] D2 h1)) ([h] E1 h) E1'
        <- ({h2:hyp A2 W}
            cut (A1 => A2) (=>R ([h1] D2 h1))
                           ([h] E2 h h2) (E2' h2))
        <- cut A1 E1' ([h1] D2 h1) F1
        <- cut A2 F1 ([h2] E2' h2) F.

c_! : cut (! A1) (!R ([o] D1 o)) ([h] !L ([h1] E1 h h1) h) F
       <- ({h1:hyp A1 W'}
            cut (! A1) (!R ([o] D1 o)) ([h] E1 h h1) (E1' h1))
       <- cut A1 (D1 W') ([h1] E1' h1) F.

c_? : cut (? A1) (?R D1) ([h] ?L ([o][h1] E1 h o h1) h) F
       <- ({o:world} {h1:hyp A1 o}
            cut (? A1) (?R D1) ([h] E1 h o h1) (E1' o h1))
       <- cut A1 D1 ([h1] E1' W' h1) F.

c_at : cut (A at W) (atR D) ([h] atL ([h'] E1 h h') h) F
        <- ({h'} cut (A at W) (atR D) ([h] E1 h h') (E1' h'))
        <- cut A D E1' F.

c_& : cut (A & B) (&R Da Db) ([h] &L ([ha][hb] E h ha hb) h) F
```

```
        <- ({ha}{hb}
             cut (A & B) (&R Da Db) ([h] E h ha hb) (E' ha hb))
        <- ({hb} cut A Da ([ha] E' ha hb) (E'' hb))
        <- cut B Db ([hb] E'' hb) F.

c_|1 : cut (A | B) (|R1 Da)
           ([h] |L ([ha] Ea h ha) ([hb] Eb h hb) h) F
    <- ({ha} cut (A | B) (|R1 Da) ([h] Ea h ha) (Fa ha))
    <- cut A Da Fa F.
c_|2 : cut (A | B) (|R2 Db)
           ([h] |L ([ha] Ea h ha) ([hb] Eb h hb) h) F
    <- ({hb} cut (A | B) (|R2 Db) ([h] Eb h hb) (Fb hb))
    <- cut B Db Fb F.

% Right commuting cuts
cr_init : cut A D ([h] init H) (init H).
cr_=>R : cut A D ([h] =>R ([h1] E1 h h1)) (=>R ([h1] F1 h1))
          <- ({h1:hyp C1 U} cut A D ([h] E1 h h1) (F1 h1)).
cr_=>L : cut A D ([h] =>L (E1 h) ([h2] E2 h h2) H)
                 (=>L F1 ([h2] F2 h2) H)
          <- cut A D ([h] E1 h) F1
          <- ({h2:hyp B2 U'} cut A D ([h] E2 h h2) (F2 h2)).
cr_!R : cut A D ([h] !R ([o] E1 h o)) (!R ([o] F1 o))
         <- ({o:world} cut A D ([h] E1 h o) (F1 o)).
cr_!L : cut A D ([h] !L ([h1] E1 h h1) H) (!L ([h1] F1 h1) H)
         <- ({h1:hyp B1 U'} cut A D ([h] E1 h h1) (F1 h1)).
cr_?R : cut A D ([h] ?R (E1 h)) (?R F1)
         <- cut A D ([h] E1 h) F1.
cr_?L : cut A D ([h] ?L ([o][h1] E1 o h1 h) H)
                (?L ([o] [h1] F1 o h1) H)
         <- ({o:world} {h1:hyp B1 o}
              cut A D ([h] E1 o h1 h) (F1 o h1)).
cr_atR : cut A D ([h] atR (E1 h)) (atR F1)
         <- cut A D E1 F1.
cr_atL : cut A D ([h] atL ([h'] E1 h' h) H) (atL F1 H)
         <- ({h'} cut A D ([h] E1 h' h) (F1 h')).
cr_&R : cut A D ([h] &R (E1 h) (E2 h)) (&R F1 F2)
         <- cut A D E1 F1
         <- cut A D E2 F2.
cr_&L : cut A D ([h] &L ([h1][h2] E h1 h2 h) H) (&L F H)
         <- ({h1}{h2} cut A D ([h] E h1 h2 h) (F h1 h2)).

cr_|R1 : cut A D ([h] |R1 (E h)) (|R1 F)
     <- cut A D E F.
cr_|R2 : cut A D ([h] |R2 (E h)) (|R2 F)
     <- cut A D E F.
cr_|L : cut A D ([h] |L ([ha] EA h ha) ([hb] EB h hb) H)
                (|L FA FB H)
     <- ({ha} cut A D ([h] EA h ha) (FA ha))
     <- ({hb} cut A D ([h] EB h hb) (FB hb)).

% Left commuting cuts
cl_=>L : cut A (=>L D1 ([h2] D2 h2) H) ([h] E h)
               (=>L D1 ([h2] F2 h2) H)
          <- ({h2:hyp B2 U'} cut A (D2 h2) ([h] E h) (F2 h2)).
cl_!L : cut A (!L ([h1] D1 h1) H) ([h] E h) (!L ([h1] F1 h1) H)
         <- ({h1:hyp B1 U'} cut A (D1 h1) ([h] E h) (F1 h1)).
cl_?L : cut A (?L ([o][h1] D1 o h1) H) ([h] E h)
              (?L ([o][h1] F1 o h1) H)
         <- ({o:world} {h1:hyp B1 o}
               cut A (D1 o h1) ([h] E h) (F1 o h1)).
cl_atL : cut A (atL ([h'] D1 h') H) ([h] E h) (atL ([h'] F1 h') H)
         <- ({h'} cut A (D1 h') ([h] E h) (F1 h')).
cl_&L : cut A (&L ([h1][h2] D h1 h2) H) ([h] E h)
              (&L ([h1][h2] F h1 h2) H)
         <- ({h1}{h2} cut A (D h1 h2) ([h] E h) (F h1 h2)).
```

```
cl_|L : cut A (|L Da Db H) E (|L Fa Fb H)
     <- ({ha} cut A (Da ha) E (Fa ha))
     <- ({hb} cut A (Db hb) E (Fb hb)).


%block lo : block {o:world}.
%block lh : some {A:prop} {W:world} block {h:hyp A W}.
%worlds (lo | lh) (cut A D E F) (stripmh _ _) (exp _ _ _) (shift _ _ _).
%total M (exp M _ _).
%total D (shift _ D _).
%total M (stripmh M _).
%total {A {D E}} (cut A D E F).


%%% Translation of ND to SEQ

ndseq : A @ W -> conc A W -> type.      %name ndseq R r.
%mode ndseq +N -D.

% Structural
ns_get : ndseq (get MOB N : A @ W) (F : conc A W)
      <- ndseq N (D : conc A W')
      <- shift MOB D F.

% Disjunction
ns_|I1 : ndseq (|I1 N) (|R1 D)
      <- ndseq N D.

ns_|I2 : ndseq (|I2 N) (|R2 D)
      <- ndseq N D.

ns_|E : ndseq (|E D DA DB) F
     <- ndseq D D'
     <- ({x : A @ W}{ha}{_:ndseq x (init ha)}
           ndseq (DA x) (DA' ha))
     <- ({x : B @ W}{hb}{_:ndseq x (init hb)}
           ndseq (DB x) (DB' hb))
     <- cut (A | B) D' ([h] |L DA' DB' h) F.

% Implication
ns_=>I : ndseq (=>I ([u1] N2 u1)) (=>R ([h1] D2 h1))
          <- ({u1:A1 @ W} {h1:hyp A1 W}
                ndseq u1 (init h1) -> ndseq (N2 u1) (D2 h1)).
ns_=>E : ndseq (=>E N2 N1) D
          <- ndseq N2 D'
          <- ndseq N1 D1
          <- cut (A1 => A2) D' ([h] =>L D1 ([h2] init h2) h) D.

% Necessity
ns_!I : ndseq (!I ([o] N1 o)) (!R ([o] D1 o))
         <- ({o:world} ndseq (N1 o) (D1 o)).
ns_!E : ndseq (!E N1) D
         <- ndseq N1 D'
         <- cut (! A1) D' ([h] !L ([h1] init h1) h) D.

% Possibility
ns_?I : ndseq (?I N1) (?R D1)
         <- ndseq N1 D1.
ns_?E : ndseq (?E N1 ([o][u1] N2 o u1)) D
         <- ndseq N1 D1'
         <- ({o:world} {u1:A1 @ o} {h1:hyp A1 o}
               ndseq u1 (init h1) -> ndseq (N2 o u1) (D2 o h1))
         <- cut (? A1) D1' ([h] ?L ([o][h1] D2 o h1) h) D.

% Hybrid
ns_atI : ndseq (atI N) (atR D)
```

```
         <- ndseq N D.
ns_atE : ndseq (atE N1 ([u] N2 u)) D
         <- ndseq N1 D1'
         <- ({u}{h} ndseq u (init h) -> ndseq (N2 u) (D2 h))
         <- cut (A at W) D1' ([h] atL ([h'] D2 h') h) D.

% Conjunction
ns_&I : ndseq (&I N1 N2) (&R D1 D2)
         <- ndseq N1 D1
         <- ndseq N2 D2.
ns_&E1 : ndseq (&E1 (N : A & B @ W)) F
         <- ndseq N D
         <- cut (A & B) D ([h] &L ([ha][hb] init ha) h) F.
ns_&E2 : ndseq (&E2 (N : A & B @ W)) F
         <- ndseq N D
         <- cut (A & B) D ([h] &L ([ha][hb] init hb) h) F.

%block luhs : some {A:prop} {W:world}
              block {u:A @ W} {h:hyp A W} {r:ndseq u (init h)}.

%worlds (lo | luhs) (ndseq N D).
%total N (ndseq N D).

%%% Translation of SEQ to ND

% Another strengthening lemma, trivial...
eraseh : (hyp A W -> B @ W') -> B @ W' -> type.
%name eraseh E e.
%mode eraseh +D -D'.

- : eraseh ([x] &I (D1 x) (D2 x)) (&I D1' D2')
 <- eraseh D1 D1'
 <- eraseh D2 D2'.

% covers some var cases that can't occur
- : eraseh ([x] D) D.

- : eraseh ([x] |E (D x) ([a] Da x a) ([b] Db x b))
           (|E D' Da' Db')
 <- eraseh D D'
 <- ({a} eraseh ([x] Da x a) (Da' a))
 <- ({b} eraseh ([x] Db x b) (Db' b)).

- : eraseh ([x] |I1 (D x)) (|I1 D') <- eraseh D D'.
- : eraseh ([x] |I2 (D x)) (|I2 D') <- eraseh D D'.

- : eraseh ([x] &E1 (D x)) (&E1 D') <- eraseh D D'.
- : eraseh ([x] &E2 (D x)) (&E2 D') <- eraseh D D'.

- : eraseh ([x] get (M x) (D x)) (get M' D')
% luckily we already proved this too
 <- stripmh M M'
 <- eraseh D D'.

- : eraseh ([x] atI (D x)) (atI D') <- eraseh D D'.

- : eraseh ([x] ?E (D x) ([o][a] D2 o a x)) (?E D' D2')
 <- eraseh D D'
 <- ({o}{a} eraseh ([x] D2 o a x) (D2' o a)).

- : eraseh ([x] ?I (D x)) (?I D') <- eraseh D D'.

- : eraseh ([x] atE (D x) ([a] D2 a x)) (atE D' D2')
 <- eraseh D D'
 <- ({a} eraseh ([x] D2 a x) (D2' a)).
```

```
- : eraseh ([x] !E (D x)) (!E D') <- eraseh D D'.

- : eraseh ([x] !I ([w] D x w)) (!I D')
 <- ({w} eraseh ([x] D x w) (D' w)).

- : eraseh ([x] =>E (D1 x) (D2 x)) (=>E D1' D2')
 <- eraseh D1 D1'
 <- eraseh D2 D2'.

- : eraseh ([x] =>I ([y] D x y)) (=>I D')
 <- ({y} eraseh ([x] D x y) (D' y)).

seqnd : conc A W -> A @ W -> type.      %name seqnd S.
hypnd : hyp A W -> A @ W -> type.       %name hypnd T t.
%mode seqnd +D -N.
%mode hypnd +H -N.

% Init
sn_init : seqnd (init H) N <- hypnd H N.

% Necessity
sn_!R : seqnd (!R ([o] D1 o)) (!I ([o] N1 o))
         <- ({o:world} seqnd (D1 o) (N1 o)).
sn_!L : seqnd (!L ([h1] D2 h1) H) (N2 (!E (!G N)))
         <- ({h1:hyp A1 W'} {u1:A1 @ W'}
               hypnd h1 u1 -> seqnd (D2 h1) (N2' h1 u1))
         <- ({u} eraseh ([h] N2' h u) (N2 u))
         <- hypnd H N.

% Possibility
sn_?R : seqnd (?R D1) (?G (?I N1)) <- seqnd D1 N1.
sn_?L : seqnd (?L ([o][h1] D2 o h1) H)
              (?E (?G N) ([o] [u1] N2 o u1))
     <- ({o:world} {h1:hyp A1 o} {u1:A1 @ o}
   hypnd h1 u1 -> seqnd (D2 o h1) (N2' o h1 u1))
     <- ({o}{u} eraseh ([h] N2' o h u) (N2 o u))
     <- hypnd H N.

% Hybrid
sn_atR : seqnd (atR D : conc (A at W') W)
               (get atmob (atI N : (A at W') @ W'))
      <- seqnd (D : conc A W') (N : A @ W').
sn_atL : seqnd (atL ([h] D h) H) (atE (atG N1) ([u] N2 u))
      <- ({h : hyp A W}{u : A @ W}
    hypnd h u -> seqnd (D h) (N2' h u))
      <- ({u} eraseh ([h] N2' h u) (N2 u))
      <- hypnd H N1.

% Conjunction
sn_&R : seqnd (&R D1 D2) (&I N1 N2)
     <- seqnd D1 N1
     <- seqnd D2 N2.
sn_&L : seqnd (&L ([ha][hb] D ha hb) H) (N (&E1 Nab) (&E2 Nab))
     <- ({ha : hyp A W}{ua : A @ W}{t : hypnd ha ua}
 {hb : hyp B W}{ub : B @ W}{t : hypnd hb ub}
   seqnd (D ha hb) (N'' ha hb ua ub))
     <- ({ha}{ua}{_:hypnd ha ua}{ub}
   eraseh ([hb] N'' ha hb ua ub) (N' ha ua ub))
     <- ({ua}{ub} eraseh ([ha] N' ha ua ub) (N ua ub))
     <- hypnd H Nab.

% Implication
sn_=>R : seqnd (=>R ([h1] D2 h1)) (=>I ([u1] N2 u1))
      <- ({h1:hyp A1 W} {u1:A1 @ W} hypnd h1 u1 ->
   seqnd (D2 h1) (N2' h1 u1))
      <- ({u} eraseh ([h] N2' h u) (N2 u)).
```

```
sn_=>L : seqnd (=>L D1 ([h2] D2 h2) H) (N2 (=>E N N1))
      <- seqnd D1 N1
      <- ({h2:hyp A2 W} {u2:A2 @ W} hypnd h2 u2 ->
     seqnd (D2 h2) (N2' h2 u2))
      <- ({u} eraseh ([h] N2' h u) (N2 u))
      <- hypnd H N.

% Disjunction
sn_|R1 : seqnd (|R1 D) (|I1 N) <- seqnd D N.
sn_|R2 : seqnd (|R2 D) (|I2 N) <- seqnd D N.

% uses the derived form |Er
sn_|L : seqnd (|L Da Db H) (|Er Nab Na Nb)
     <- ({ha}{a}{_ : hypnd ha a} seqnd (Da ha) (Na' ha a))
     <- ({hb}{b}{_ : hypnd hb b} seqnd (Db hb) (Nb' hb b))
     <- ({a} eraseh ([h] Na' h a) (Na a))
     <- ({b} eraseh ([h] Nb' h b) (Nb b))
     <- hypnd H Nab.

%block lhut : some {A:prop} {W:world}
              block {h:hyp A W} {u:A @ W} {t:hypnd h u}.
%block lx : some {A:prop}{W:world} block {_ : A @ W}.
%worlds (lo | lhut | lx) (seqnd D N) (hypnd H N') (eraseh _ _).
%total D (eraseh D _).
%total (D H) (seqnd D N) (hypnd H N').
```

A.2 Equivalence of IS5$^{\cup}$ natural deduction and sequent calculus

This signature requires the signature from Appendix A.1 as a prefix.

```
%% Equivalence of IS5U and IS5U sequent calculus

%%% Based on proofs by Frank Pfenning, 2004
%%% Revised and extended by Tom Murphy VII, 2007

% IS5U Natural deduction

@@ : prop -> world -> type.              %name @@ N.
%infix none 1 @@.

% Implication
==>I : (A @@ W -> B @@ W) -> (A => B @@ W).
==>E : (A => B @@ W) -> A @@ W -> B @@ W.

% Necessity
!!I : ({o:world} A @@ o) -> ! A @@ W.
!!E : ! A @@ W -> A @@ W'.

% Possibility
??I : A @@ W -> ? A @@ W'.
??E : ? A @@ W' -> ({o:world} A @@ o -> C @@ W) -> C @@ W.

% Hybrid
attI : A @@ W -> A at W @@ W'.
attE : A at W' @@ W'' -> (A @@ W' -> C @@ W) -> C @@ W.

% Conjunction
&&I : A @@ W -> B @@ W -> A & B @@ W.
&&E1 : A & B @@ W -> A @@ W.
&&E2 : A & B @@ W -> B @@ W.
```

```
% Disjunction
||I1 : A @@ W -> A | B @@ W.
||I2 : B @@ W -> A | B @@ W.
||E : A | B @@ W' -> (A @@ W' -> C @@ W) -> (B @@ W' -> C @@ W) -> C @@ W.

%%% Translation of ND to SEQ

nndseq : A @@ W -> conc A W -> type.      %name nndseq R r.
%mode nndseq +N -D.

% Disjunction
- : nndseq (||I1 N) (|R1 D) <- nndseq N D.
- : nndseq (||I2 N) (|R2 D) <- nndseq N D.
- : nndseq (||E D DA DB) F
 <- nndseq D D'
 <- ({x : A @@ W}{ha}{_:nndseq x (init ha)}
       nndseq (DA x) (DA' ha))
 <- ({x : B @@ W}{hb}{_:nndseq x (init hb)}
       nndseq (DB x) (DB' hb))
 <- cut (A | B) D' ([h] |L DA' DB' h) F.

% Implication
- : nndseq (==>I ([u1] N2 u1)) (=>R ([h1] D2 h1))
 <- ({u1:A1 @@ W} {h1:hyp A1 W}
               nndseq u1 (init h1) -> nndseq (N2 u1) (D2 h1)).
- : nndseq (==>E N2 N1) D
 <- nndseq N2 D'
 <- nndseq N1 D1
 <- cut (A1 => A2) D' ([h] =>L D1 ([h2] init h2) h) D.

% Necessity
- : nndseq (!!I ([o] N1 o)) (!R ([o] D1 o))
 <- ({o:world} nndseq (N1 o) (D1 o)).
- : nndseq (!!E N1) D
 <- nndseq N1 D'
 <- cut (! A1) D' ([h] !L ([h1] init h1) h) D.

% Possibility
- : nndseq (??I N1) (?R D1)
 <- nndseq N1 D1.
- : nndseq (??E N1 ([o][u1] N2 o u1)) D
 <- nndseq N1 D1'
 <- ({o:world} {u1:A1 @@ o} {h1:hyp A1 o}
       nndseq u1 (init h1) -> nndseq (N2 o u1) (D2 o h1))
 <- cut (? A1) D1' ([h] ?L ([o][h1] D2 o h1) h) D.

% Hybrid
- : nndseq (attI N) (atR D)
 <- nndseq N D.
- : nndseq (attE N1 ([u] N2 u)) D
 <- nndseq N1 D1'
 <- ({u}{h} nndseq u (init h) -> nndseq (N2 u) (D2 h))
 <- cut (A at W) D1' ([h] atL ([h'] D2 h') h) D.

% Conjunction
- : nndseq (&&I N1 N2) (&R D1 D2)
 <- nndseq N1 D1
 <- nndseq N2 D2.
- : nndseq (&&E1 (N : A & B @@ W)) F
 <- nndseq N D
 <- cut (A & B) D ([h] &L ([ha][hb] init ha) h) F.
- : nndseq (&&E2 (N : A & B @@ W)) F
 <- nndseq N D
 <- cut (A & B) D ([h] &L ([ha][hb] init hb) h) F.

%block luhs : some {A:prop} {W:world}
```

```
              block {u:A @@ W} {h:hyp A W} {r:nndseq u (init h)}.

%worlds (lo | luhs) (nndseq N D).
%total N (nndseq N D).

%%% Translation of SEQ to ND

sseqnd : conc A W -> A @@ W -> type.      %name sseqnd S.
hhypnd : hyp A W -> A @@ W -> type.       %name hhypnd T t.
%mode sseqnd +D -N.
%mode hhypnd +H -N.

% Init
- : sseqnd (init H) N <- hhypnd H N.

% Necessity
- : sseqnd (!R ([o] D1 o)) (!!I ([o] N1 o))
 <- ({o:world} sseqnd (D1 o) (N1 o)).
- : sseqnd (!L ([h1] D2 h1) H) (N2 (!!E N))
 <- ({h1:hyp A1 W'} {u1:A1 @@ W'}
       hhypnd h1 u1 -> sseqnd (D2 h1) (N2 u1))
 <- hhypnd H N.

% Possibility
- : sseqnd (?R D1) (??I N1)
 <- sseqnd D1 N1.
- : sseqnd (?L ([o][h1] D2 o h1) H)
           (??E N ([o] [u1] N2 o u1))
 <- ({o:world} {h1:hyp A1 o} {u1:A1 @@ o}
       hhypnd h1 u1 -> sseqnd (D2 o h1) (N2 o u1))
 <- hhypnd H N.

% Hybrid
- : sseqnd (atR D) (attI N) <- sseqnd D N.
- : sseqnd (atL ([h] D h) H)
           (attE N1 ([u] N2 u))
 <- ({h : hyp A W}{u : A @@ W}
       hhypnd h u -> sseqnd (D h) (N2 u))
 <- hhypnd H N1.

% Conjunction
- : sseqnd (&R D1 D2) (&&I N1 N2)
 <- sseqnd D1 N1
 <- sseqnd D2 N2.
- : sseqnd (&L ([ha][hb] D ha hb) H) (N (&&E1 Nab) (&&E2 Nab))
 <- ({ha : hyp A W}{ua : A @@ W}{t : hhypnd ha ua}
     {hb : hyp B W}{ub : B @@ W}{t : hhypnd hb ub}
       sseqnd (D ha hb) (N ua ub))
 <- hhypnd H Nab.

% Implication
- : sseqnd (=>R ([h1] D2 h1)) (==>I ([u1] N2 u1))
 <- ({h1:hyp A1 W} {u1:A1 @@ W} hhypnd h1 u1 ->
      sseqnd (D2 h1) (N2 u1)).
- : sseqnd (=>L D1 ([h2] D2 h2) H) (N2 (==>E N N1))
 <- sseqnd D1 N1
 <- ({h2:hyp A2 W} {u2:A2 @@ W} hhypnd h2 u2 ->
      sseqnd (D2 h2) (N2 u2))
 <- hhypnd H N.

% Disjunction
- : sseqnd (|R1 D) (||I1 N) <- sseqnd D N.
- : sseqnd (|R2 D) (||I2 N) <- sseqnd D N.

- : sseqnd (|L Da Db H) (||E Nab Na Nb)
 <- ({ha}{a}{_ : hhypnd ha a} sseqnd (Da ha) (Na a))
```

```
 <- ({hb}{b}{_ : hhypnd hb b} sseqnd (Db hb) (Nb b))
 <- hhypnd H Nab.

%block lhut : some {A:prop} {W:world}
              block {h:hyp A W} {u:A @@ W} {t:hhypnd h u}.
%block lx : some {A:prop}{W:world} block {_ : A @@ W}.
%worlds (lo | lhut | lx) (sseqnd D N) (hhypnd H N').
%total (D H) (sseqnd D N) (hhypnd H N').
```

A.3 Lambda 5 dynamic semantics

A.3.1 The `%partial` extension

This proof uses an experimental extension to Twelf that adds a keyword `%partial`. It has the same syntax as `%total` [113], but supresses the termination check. This differs from `%covers` in that it performs output coverage checking, output freeness checking, and ensures that if the relation in question appeals to others, then those others are either total or partial.

This is used in the formalization of the dynamic semantics to verify that the big-step evaluation relation always makes progress: that it either terminates producing a value or results in an infinite derivation. Such an argument is coinductive on paper. The logic programming formalization gives us an elegant way to state it, however: By expressing the big-step operational semantics as a logic program, an "infinite derivation" is just a non-terminating proof search in Twelf's operational semantics. Therefore, progress in metalanguage's proof search (established by `%partial`) is the notion of progress we desire for the object language's evaluation relation.

A.3.2 Dynamic semantics

This signature stands alone.

```
% Operational semantics for Lambda 5.
% We redefine the language so that we can
% make a distinction between values and expressions.

world : type.                           %name world W w.
prop : type.                            %name prop A x.

% Propositions

=> : prop -> prop -> prop.              %infix right 8 =>.
! : prop -> prop.                       %prefix 9 !.
? : prop -> prop.                       %prefix 9 ?.
at : prop -> world -> prop.             %infix none 6 at.
& : prop -> prop -> prop.               %infix left 7 &.
| : prop -> prop -> prop.               %infix left 7 |.

% Natural deduction

mobile : prop -> type.                  %name mobile M m.
@ : prop -> world -> type.              %name @ N.
val : prop -> world -> type.            %name val V v.
%infix none 1 @.
```

```
!mob  : mobile (! A).
?mob  : mobile (? A).
atmob : mobile (A at W).
&mob  : mobile A -> mobile B -> mobile (A & B).
|mob  : mobile A -> mobile B -> mobile (A | B).

% Implication
lam : (val A W -> B @ W) -> (A => B @ W).
app : (A => B @ W) -> A @ W -> B @ W.

% Necessity
box : ({o:world} A @ o) -> ! A @ W.
unbox : ! A @ W -> A @ W.

% Possibility
here : A @ W -> ? A @ W.
letd : ? A @ W -> ({o:world} val A o -> C @ W) -> C @ W.

% Hybrid
hold : A @ W -> A at W @ W.
leta : A at W' @ W -> (val A W' -> C @ W) -> C @ W.

% Conjunction
, : A @ W -> B @ W -> A & B @ W.        %infix none 5 ,.
#1 : A & B @ W -> A @ W.
#2 : A & B @ W -> B @ W.

% Disjunction
inl : A @ W -> A | B @ W.
inr : B @ W -> A | B @ W.
case : A | B @ W -> (val A W -> C @ W) -> (val B W -> C @ W) -> C @ W.

get : mobile A -> A @ W -> A @ W'.
v : val A W -> A @ W.

% Values have slightly different typing
% rules: we allow non-local introduction forms.

vpair : val A W -> val B W -> val (A & B) W.
vinl : val A W -> val (A | B) W.
vinr : val B W -> val (A | B) W.
vhold : val A W' -> val (A at W') W.
vthere : val A W' -> val (? A) W.
vlam : (val A W -> B @ W) -> val (A => B) W.
vbox : ({w} A @ w) -> val (! A) W.

% big-step operational semantics

eval : A @ W -> val A W -> type.
%mode eval +EXP -VAL.

eval_case : val (A | B) W -> (val A W -> C @ W)
                          -> (val B W -> C @ W)
                          ->  val C W -> type.
%mode eval_case +V +N1 +N2 -V'.

vshift : mobile A -> val A W -> val A W' -> type.
%mode +{A:prop} +{M:mobile A} +{W:world} +{W':world}
       +{V:val A W} -{V':val A W'}
       vshift M V V'.

s-box : vshift !mob (vbox F) (vbox F).
s-dia : vshift ?mob (vthere V) (vthere V).
s-at  : vshift atmob (vhold V) (vhold V).
s-inl : vshift (|mob Ma _) (vinl V) (vinl V') <- vshift Ma V V'.
s-inr : vshift (|mob _ Mb) (vinr V) (vinr V') <- vshift Mb V V'.
```

```
s-pair : vshift (&mob Ma Mb) (vpair Va Vb) (vpair Va' Vb')
      <- vshift Ma Va Va'
      <- vshift Mb Vb Vb'.

ec-inl : eval_case (vinl V) N1 _ V'
      <- eval (N1 V) V'.
ec-inr : eval_case (vinr V) _ N2 V'
      <- eval (N2 V) V'.

ev-inl : eval (inl M) (vinl V)
      <- eval M V.
ev-inr : eval (inr M) (vinr V)
      <- eval M V.
ev-, : eval (M , N) (vpair V U)
    <- eval M V
    <- eval N U.
ev-#1 : eval (#1 M) V1
     <- eval M (vpair V1 V2).
ev-#2 : eval (#2 M) V2
     <- eval M (vpair V1 V2).
ev-case : eval (case M N1 N2) V'
       <- eval M V
       <- eval_case V N1 N2 V'.
ev-lam : eval (lam M) (vlam M).
ev-box : eval (box M) (vbox M).
ev-val : eval (v V) V.
ev-app : eval (app M N) V'
      <- eval M (vlam F)
      <- eval N V
      <- eval (F V) V'.
ev-leta : eval (leta M N) V'
       <- eval M (vhold V)
       <- eval (N V) V'.
ev-unbox : eval (unbox M) V'
        <- eval M (vbox F)
        <- eval (F _) V'.
ev-hold : eval (hold M) (vhold V)
       <- eval M V.
ev-letd : eval (letd M ([w][x] N w x)) V'
       <- eval M (vthere (V : val A W'))
       <- eval (N W' V) V'.
ev-here : eval (here M) (vthere V)
       <- eval M V.
ev-get : eval (get MOB M) V'
      <- eval M V
      <- vshift MOB V V'.

%block blockw : block {w:world}.
%worlds (blockw) (eval _ _) (eval_case _ _ _ _) (vshift _ _ _).
%total M (vshift M V _).
%covers eval_case +V +N1 +N2 -V'.
%covers eval +D -V.
% experimental feature
%partial (M N) (eval_case M _ _ _) (eval N _).
```

A.4 Soundness and completeness of C5

This signature stands alone.

```
%% Classical Judgmental S5
%% Tom Murphy VII
%% Thanks: Frank Pfenning and Jason Reed
%%  8 Apr 2004
```

```
world : type.                          %name world W w.
prop : type.                           %name prop A.


%%% Natural deduction

% truth assumptions and conclusions
@ : prop -> world -> type.             %name @ N.
%infix none 1 @.
% falsehood (continuation) assumptions
* : prop -> world -> type.             %name * X.
%infix none 1 *.

=> : prop -> prop -> prop.             %infix right 8 =>.
! : prop -> prop.                      %prefix 9 !.
? : prop -> prop.                      %prefix 9 ?.
& : prop -> prop -> prop.              %infix none 9 &.
at : prop -> world -> prop.            %infix none 3 at.

mobile : prop -> type.                 %name mobile M m.
!mob  : mobile (! A).
?mob  : mobile (? A).
atmob : mobile (A at W).
&mob  : mobile A -> mobile B -> mobile (A & B).

% Structural Rules

get : mobile A -> A @ W -> A @ W'.
letcc : (A * W -> A @ W) -> A @ W.
throw : A @ W -> (A * W -> C @ W').

% Implication
=>I : (A @ W -> B @ W) -> (A => B @ W).
=>E : (A => B @ W) -> A @ W -> B @ W.

% Hybrid
atI : A @ W -> (A at W @ W).
atE : A at W' @ W -> (A @ W' -> C @ W'') -> C @ W''.

% Necessity
!I : ({o:world} A @ o) -> ! A @ W.
!E : ! A @ W -> A @ W.
!G : ! A @ W' -> ! A @ W = [m] get !mob m.

% Possibility
?I : A @ W -> ? A @ W.
?E : ? A @ W -> ({o:world} A @ o -> C @ W) -> C @ W.
?G : ? A @ W' -> ? A @ W = [m] get ?mob m.

% Conjunction
&I : A @ W -> B @ W -> (A & B @ W).
&E1 : (A & B @ W) -> A @ W.
&E2 : (A & B @ W) -> B @ W.

% Falsehood
_|_ : prop.
_|_E : _|_ @ W -> C @ W'.


%%% Sequent calculus

true  : prop -> world -> type.       %name true T t.
false : prop -> world -> type.       %name false F f.

# : type.
```

```
% judgmental
contra : true A W -> false A W -> #.

% hybrid
atT : (true A W' -> #) ->
      (true (A at W') W -> #).
atF : (false A W' -> #) ->
      (false (A at W') W -> #).

% arrow
=>T : (false A W -> #)  ->  (true B W -> #)  ->
            (true (A => B) W -> #).
=>F : (true A W -> false B W -> #) ->
        (false (A => B) W -> #).

% box
!T : (true A W' -> #) ->
     (true (! A) W -> #).
!F : ({w:world} false A w -> #) ->
     (false (! A) W -> #).

% dia
?T : ({w:world} true A w -> #) ->
     (true (? A) W -> #).
?F : (false A W' -> #) ->
     (false (? A) W -> #).

% conjunction
&T : (true A W -> true B W -> #) ->
        (true (A & B) W -> #).
&F : (false A W -> #)   ->   (false B W -> #) ->
        (false (A & B) W -> #).

% falsehood

_|_T : true _|_ W -> #.

% admissibility of excluded middle (cut)

xm : {A:prop} {W:world} (true A W -> #) -> (false A W -> #) -> # -> type.
%mode xm +A +W +D +E -F.

% initial cuts
xt_init : xm A W ([xt] contra xt D) ([xf] E xf) (E D).
xf_init : xm A W ([xt] D xt) ([xf] contra E xf) (D E).

% unused assumptions
xt_unused : xm A W ([x1] contra DT DF) E (contra DT DF).
xf_unused : xm A W D ([x1] contra ET EF) (contra ET EF).

% falsehood. actually falsehood is trivial because a principal
% cut is impossible, and the "commutative" cases aren't even inductive.
xd_ct_|_ : xm A W ([a] _|_T DD) _ (_|_T DD).
xe_ct_|_ : xm A W _ ([af] _|_T EE) (_|_T EE).

% commutative cases.
% note that we need to consider each rule in both D and E, so
% there are four cases for each connective.

% implication
xd_ct=> : xm A W ([a : true A W] =>T
                   ([cf : false C W'] D1 a cf)
                   ([dt : true D W'] D2 a dt) DD) ([af] E af)
               (=>T ([cf : false C W'] F1 cf) ([dt : true D W'] F2 dt) DD)
       <- ({cf : false C W'} xm A W ([a] D1 a cf) E (F1 cf))
       <- ({dt : true D W'}  xm A W ([a] D2 a dt) E (F2 dt)).
```

```
xe_ct=> : xm A W ([a] DD a) ([af : false A W] =>T
                               ([cf : false C W'] E1 af cf)
                               ([dt : true D W'] E2 af dt) EE)
              (=>T ([cf] F1 cf) ([dt] F2 dt) EE)
       <- ({cf : false C W'} xm A W DD ([af] E1 af cf) (F1 cf))
       <- ({dt : true D W'}  xm A W DD ([af] E2 af dt) (F2 dt)).

xd_cf=> : xm A W ([a : true A W] =>F
                  ([ct : true C W'] [df : false D W'] D' a ct df) DD)
              ([af : false A W] E af)
              (=>F ([ct : true C W'][df : false D W'] F ct df) DD)
       <- ({ct : true C W'} {df : false D W'}
             xm A W ([a] D' a ct df) E (F ct df)).

xe_cf=> : xm A W ([a : true A W] DD a)
              ([af : false A W] =>F
                 ([ct : true C W'] [df : false D W'] E' af ct df) EE)
              (=>F ([ct : true C W'][df : false D W'] F ct df) EE)
       <- ({ct : true C W'} {df : false D W'}
             xm A W DD ([af] E' af ct df) (F ct df)).

% box
xd_ct! : xm A W ([a : true A W] !T ([bt : true B W'] D1 a bt) DD)
                ([af] E af)
                (!T ([bt : true B W'] F1 bt) DD)
      <- ({bt : true B W'} xm A W ([a] D1 a bt) E (F1 bt)).

xe_ct! : xm A W ([a] D a)
                ([af : false A W] !T ([bt : true B W'] E1 af bt) EE)
                (!T ([bt : true B W'] F1 bt) EE)
      <- ({bt : true B W'} xm A W D ([af] E1 af bt) (F1 bt)).

xd_cf! : xm A W ([a : true A W] !F ([w][bf : false B w] D1 a w bf) DD)
                ([af] E af)
                (!F ([w][bf] F1 w bf) DD)
      <- ({w : world}{bf : false B w}
            xm A W ([a] D1 a w bf) E (F1 w bf)).

xe_cf! : xm A W ([a] D a)
                ([af] !F ([w][bf : false B w] E1 af w bf) EE)
                (!F ([w][bf] F1 w bf) EE)
      <- ({w}{bf} xm A W D ([af] E1 af w bf) (F1 w bf)).

% dia
xd_ct? : xm A W ([a : true A W] ?T ([w][bt] D1 a w bt) DD)
                ([af] E af)
                (?T ([w][bt] F1 w bt) DD)
      <- ({w}{bt} xm A W ([a] D1 a w bt) ([af] E af) (F1 w bt)).

xe_ct? : xm A W ([a] D a)
                ([af] ?T ([w][bt] E1 af w bt) EE)
                (?T ([w][bt] F1 w bt) EE)
      <- ({w}{bt} xm A W ([a] D a) ([af] E1 af w bt) (F1 w bt)).

xd_cf? : xm A W ([a] ?F ([bf : false B W'] D1 a bf) DD)
                ([af] E af)
                (?F ([bf] F1 bf) DD)
      <- ({bf} xm A W ([a] D1 a bf) ([af] E af) (F1 bf)).

xe_cf? : xm A W ([a] D a)
                ([af] ?F ([bf : false B W'] E1 af bf) EE)
                (?F ([bf] F1 bf) EE)
      <- ({bf} xm A W ([a] D a) ([af] E1 af bf) (F1 bf)).

% hybrid
```

```
xd_ctat : xm A W ([a] atT ([bt] D a bt) DD)
                 ([af] E af)
                 (atT ([bt] F bt) DD)
       <- ({bt} xm A W ([a] D a bt) ([af] E af) (F bt)).
xe_ctat : xm A W ([a] D a)
                 ([af] atT ([bt] E af bt) EE)
                 (atT ([bt] F bt) EE)
       <- ({bt} xm A W D ([af] E af bt) (F bt)).
xd_cfat : xm A W ([a] atF ([bf] D a bf) DD) E
                 (atF ([bf] F bf) DD)
       <- ({bf} xm A W ([a] D a bf) E (F bf)).
xe_cfat : xm A W D ([af] atF ([bf] E af bf) EE)
                 (atF ([bf] F bf) EE)
       <- ({bf} xm A W D ([af] E af bf) (F bf)).

% conjunction
xd_ct& : xm A W ([a] &T ([ct][dt] D1 a ct dt) DD)
                ([af] E af)
                (&T ([ct][dt] F1 ct dt) DD)
      <- ({ct}{dt} xm A W ([a] D1 a ct dt) ([af] E af) (F1 ct dt)).

xe_ct& : xm A W ([a] D a)
                ([af] &T ([ct][dt] E1 af ct dt) EE)
                (&T ([ct][dt] F1 ct dt) EE)
      <- ({ct}{dt} xm A W ([a] D a) ([af] E1 af ct dt) (F1 ct dt)).

xd_cf& : xm A W ([a] &F ([cf] D1 a cf) ([df] D2 a df) DD)
                ([af] E af)
                (&F ([cf] F1 cf) ([df] F2 df) DD)
      <- ({cf} xm A W ([a] D1 a cf) ([af] E af) (F1 cf))
      <- ({df} xm A W ([a] D2 a df) ([af] E af) (F2 df)).

xe_cf& : xm A W ([a] D a)
                ([af] &F ([cf] E1 af cf) ([df] E2 af df) EE)
                (&F ([cf] F1 cf) ([df] F2 df) EE)
      <- ({cf} xm A W ([a] D a) ([af] E1 af cf) (F1 cf))
      <- ({df} xm A W ([a] D a) ([af] E2 af df) (F2 df)).


% principal cuts. there is one case for each connective;
% a use of the true  rule on A in D, and
% a use of the false rule on A in E.

% implication
x_=> : xm (A => B) W
          ([it : true  (A => B) W] =>T ([af : false A W] D1 it af)
                                       ([bt : true  B W] D2 it bt) it)
          ([if : false (A => B) W] =>F ([a : true  A W]
                                        [bf : false B W] E1 if a bf) if)
         F
    <- ({a : true A W} {bf : false B W}
        xm (A => B) W ([it] =>T ([af : false A W] D1 it af)
                                ([bt] D2 it bt) it)
                      ([if] E1 if a bf) (E1' a bf))
    <- ({af : false A W}
        xm (A => B) W ([it] D1 it af)
                      ([if] =>F ([a][bf] E1 if a bf) if) (D1' af))
    <- ({bt : true B W}
        xm (A => B) W ([it] D2 it bt)
                      ([if] =>F ([a][bf] E1 if a bf) if) (D2' bt))
    <- ({bf : false B W}
        xm A W ([a : true A W] E1' a bf)
               ([af : false A W] D1' af) (F' bf))
    <- xm B W ([bt] D2' bt) ([bf] F' bf) F.

% box
```

```
x_! : xm (! A) W ([nt] !T ([a : true A W'] D1 nt a) nt)
                 ([nf] !F ([w][af : false A w] E1 nf w af) nf)
                 F
   <- ({w : world}{af : false A w}
       xm (! A) W ([nt] !T ([a] D1 nt a) nt)
                  ([nf] E1 nf w af) (E1' w af))
   <- ({a : true A W'}
       xm (! A) W ([nt] D1 nt a)
                  ([nf] !F ([w][af] E1 nf w af) nf) (D1' a))
   <- xm A W' ([a : true A W'] D1' a)
              ([af : false A W'] E1' W' af) F.

% dia
x_? : xm (? A) W ([nt] ?T ([w][a] D1 nt w a) nt)
                 ([nf] ?F ([af] E1 nf af) nf)
                 F
   <- ({w : world}{a : true A w}
       xm (? A) W ([nt] D1 nt w a)
                  ([nf] ?F ([af] E1 nf af) nf) (D1' w a))
   <- ({af : false A W'}
       xm (? A) W ([nt] ?T ([w][a] D1 nt w a) nt)
                  ([nf] E1 nf af) (E1' af))
   <- xm A W' ([a : true A W'] D1' W' a) ([af] E1' af) F.

% conjunction
x_& : xm (A & B) W ([&t] &T ([a][bt] D1 &t a bt) &t)
                   ([&f] &F ([af] E1 &f af) ([bf] E2 &f bf) &f)
                   F
   <- ({a}{bt} xm (A & B) W ([&t] D1 &t a bt)
                            ([&f] &F ([af] E1 &f af)
                                     ([bf] E2 &f bf) &f)
                            (D1' a bt))
   <- ({af} xm (A & B) W ([&t] &T ([a][bt] D1 &t a bt) &t)
                         ([&f] E1 &f af)
                         (E1' af))
   <- ({bf} xm (A & B) W ([&t] &T ([a][bt] D1 &t a bt) &t)
                         ([&f] E2 &f bf)
                         (E2' bf))
   <- ({bt} xm A W ([a] D1' a bt) ([af] E1' af) (D1'' bt))
   <- (xm B W ([bt] D1'' bt) ([bf] E2' bf) F).

x_at : xm (A at W') W ([att] atT ([a]  D att a)  att)
                      ([atf] atF ([af] E atf af) atf) F
    <- ({a}  xm (A at W') W ([att] D att a) ([atf] atF ([af] E atf af) atf) (DD' a))
    <- ({af} xm (A at W') W ([att] atT ([a] D att a) att) ([atf] E atf af) (EE' af))
    <- xm A W' DD' EE' F.

%block blockw : block {w : world}.
%block blockt : some {A : prop} {W : world} block {tt : true  A W}.
%block blockf : some {A : prop} {W : world} block {ff : false A W}.

%worlds (blockw | blockt | blockf) (xm A W D E F).
%total {A [D E]} (xm A W D E F).

%%% Translation from natural deduction to sequent calculus.

% G;D |- A @ W    then   G # D,A
ndseq : A @ W -> (false A W -> #) -> type.
%mode ndseq +D -F.

switch : mobile A -> (false A W -> #) -> {W':world} (false A W' -> #) -> type.
%mode switch +M +FW +W' -FW'.

- : switch !mob F1 _ F
 <- ({!f} xm (! A) W'
       ([!t' : true (! A) W']
```

```
          !F ([w'' : world][af'' : false A w'']
              (!T ([at'' : true A w''] contra at'' af'') !t')) !f)
       ([!f' : false (! A) W'] F1 !f')
       (F !f)).

- : switch ?mob F1 _ F
 <- ({?f : false (? A) W}
       xm (? A) W'
       ([?t'] ?T ([w''][at''] ?F ([af''] contra at'' af'') ?f) ?t')
       ([?f'] F1 ?f')
       (F ?f)).

- : switch atmob F1 _ F
 <- ({atf}
       xm (A at _) W'
       ([t'] atT ([at''] atF ([af''] contra at'' af'') atf) t')
       ([f'] F1 f')
       (F atf)).

- : switch (&mob Ma Mb) ([fab:false (A & B) W] F1 fab) W' ([fab':false (A & B) W']
                                                            &F Fa' Fb' fab')
 <- ({fa : false A W}
       xm (A & B) W
       ([tab : true (A & B) W] &T ([ta][tb] contra ta fa) tab)
       ([fab] F1 fab)
       (Fa fa))
 <- switch Ma Fa W' Fa'
 <- ({fb : false B W}
       xm (A & B) W
       ([tab : true (A & B) W] &T ([ta][tb] contra tb fb) tab)
       ([fab] F1 fab)
       (Fb fb))
 <- switch Mb Fb W' Fb'.

% get
ns-!G : ndseq (get M (D : A @ W')) ([f: false A W] F f)
     <- ndseq D ([!f'] F1 !f')
     <- switch M F1 W F.

% hybrid
ns-atI : ndseq (atI N) (atF F) <- ndseq N F.
ns-atE : ndseq (atE D1 ([a] D2 a))
               ([cf] F cf)
     <- ndseq D1 F1
     <- ({aa}{a}
           ndseq aa ([af] contra a af) ->
           ndseq (D2 aa) ([cf] F2 a cf))
     <- ({cf}
           xm (A at W') W ([t] atT ([a] F2 a cf) t)
                          ([f] F1 f) (F cf)).

% conjunction
ns-&I : ndseq (&I D1 D2) (&F F1 F2)
     <- ndseq D1 F1
     <- ndseq D2 F2.

ns-&E1 : ndseq (&E1 (D1 : A & B @ W))
          ([af] F af)
      <- ndseq D1 ([&f] F2 &f)
      <- ({af} xm (A & B) W ([&t] &T ([a][bt] contra a af) &t)
                            ([&f] F2 &f) (F af)).

ns-&E2 : ndseq (&E2 (D1 : A & B @ W))
           ([bf] F bf)
      <- ndseq D1 ([&f] F2 &f)
      <- ({bf} xm (A & B) W ([&t] &T ([a][bt] contra bt bf) &t)
```

```
                          ([&f] F2 &f) (F bf)).

% falsehood

ns-_|_E : ndseq (_|_E (D1 : _|_ @ W))
                ([af] F af)
       <- ndseq D1 ([_|_f] F2 _|_f)
       <- ({af} xm (_|_) W ([_|_t] _|_T _|_t)
                           ([_|_f] F2 _|_f) (F af)).

% implication

ns-=>I : ndseq (=>I ([a : A @ W] D a))
               ([=>f : false (A => B) W]
                =>F ([a : true A W][bf : false B W] F a bf) =>f)
      <- ({aa : A @ W}{a : true A W}
          ndseq aa ([af] contra a af) ->
          ndseq (D aa) ([bf] F a bf)).

ns-=>E : ndseq (=>E D1 D2)
               ([bf] F bf)
      <- ndseq D1 ([if] F1 if)
      <- ndseq D2 ([af] F2 af)
      <- ({bf} xm (A => B) W
            ([it] =>T ([af] F2 af) ([bt] contra bt bf) it)
            ([if] F1 if) (F bf)).

% box

ns-!I : ndseq (!I [w] D1 w) (!F [w][af : false A w] F w af)
     <- ({w} ndseq (D1 w) (F w)).

ns-!E : ndseq (!E D1) ([af] F af)
     <- ndseq D1 ([!f] F1 !f)
     <- ({af} xm (! A) W ([!t] !T ([a] contra a af) !t)
                         ([!f] F1 !f) (F af)).

% dia

ns-?I : ndseq (?I D) ([?f] ?F ([af] F af) ?f)
     <- ndseq D ([af] F af).

ns-?E : ndseq (?E D1 ([w : world][a : A @ w] D2 w a))
              ([cf : false C W] F cf)
     <- ndseq D1 ([?f] F1 ?f)
     <- ({w' : world}{aa : A @ w'}{a : true A w'}
           ndseq aa ([af] contra a af) ->
           ndseq (D2 w' aa) ([cf] F2 w' a cf))
     <- ({cf : false C W}
           xm (? A) W ([?t] ?T ([w][a] F2 w a cf) ?t)
                      ([?f] F1 ?f) (F cf)).

% we need a way of connecting nd continuation assumptions
% with sequent false assumptions
contfalse : A * W -> false A W -> type.
%mode contfalse +D -F.

% note contraction: F in output uses af twice
ns-letcc : ndseq (letcc ([ac] D ac)) ([af] F af af)
        <- ({ac : A * W} {af : false A W}
              contfalse ac af ->
              ndseq (D ac) (F af)).

% note weakening: [cf'] is unused.
ns-throw : ndseq (throw D K) ([cf'] F AF)
```

```
        <- ndseq D ([af] F af)
        <- contfalse K AF.

%block blockh : some {A:prop} {W:world}
              block {aa : A @ W} {a : true A W}
                    {_ : ndseq aa ([af] contra a af)}.

%block block* : some {A:prop} {W:world}
              block {ac : A * W} {af : false A W} {_ : contfalse ac af}.

%worlds (blockw | blockh | block*) (contfalse D F).
%worlds (blockw | blockh | block*) (ndseq D F) (switch _ _ _ _).

%total D (switch D _ _ _).
%total D (contfalse D F).
%total D (ndseq D F).

%% Continuation Substitution (excluded middle) for natural deduction
%% This is the price we pay for using A * W instead of hoas for conts

% if   G,x:A@W; D |- M : *
% and  G; D,u:A@W |- N : B
% then G; D       |- [[ x.M/u ]] N : B

xs : ({c}{w} A @ W -> c @ w) ->
     (A * W -> B @ W') ->
         (B @ W') -> type.               %name xs U.
%mode xs +D +E -F.

% special: for any term closed wrt the continuation assumption,
%   substitution is the identity. This keeps us from having to
%   treat the case of @ variables, since they are always closed
%   wrt * variables. Without this trick, world subsumption forces
%   cases of xs to infect any later theorem that uses it!

xs-closed : xs D ([a*] E) E.

xs-get : xs D ([u] get M (EE u)) (get M EE')
      <- xs D EE EE'.

% falsehood
xs-_|_E : xs D ([u] _|_E (EE u)) (_|_E EE')
       <- xs D EE EE'.
xs-atI : xs D ([u] atI (E u)) (atI F) <- xs D E F.
xs-atE : xs D ([u] (atE (E1 u) ([a] E2 a u))) (atE F1 F2)
     <- xs D E1 F1
     <- ({a'} xs D (E2 a') (F2 a')).

% conjunction
xs-&I : xs D ([u] &I (EA u) (EB u)) (&I F1 F2)
     <- xs D EA F1
     <- xs D EB F2.
xs-&E1 : xs D ([u] &E1 (E u)) (&E1 F) <- xs D E F.
xs-&E2 : xs D ([u] &E2 (E u)) (&E2 F) <- xs D E F.

% implication
xs-=>I : xs D ([u] =>I ([aw : A @ W] E aw u)) (=>I ([aw] F aw))
      <- ({aw : A @ W} xs D (E aw) (F aw)).
xs-=>E : xs D ([u] =>E (DF u) (DA u)) (=>E FF FA)
      <- xs D DF FF
      <- xs D DA FA.

% box
xs-!I : xs D ([u : A * W] !I ([w] E u w)) (!I ([w] F w))
     <- ({w : world} xs D ([u] E u w) (F w)).
xs-!G : xs D ([u] !G (E u)) (!G F) <- xs D E F.
```

```
xs-!E : xs D ([u] !E (E u)) (!E F) <- xs D E F.

% possibility
xs-?E : xs D ([u] (?E (E1 u) ([w][a] E2 w a u))) (?E F1 ([w][a] F2 w a))
     <- xs D E1 F1
     <- ({w:world}{aw: A @ w}
           xs D (E2 w aw) (F2 w aw)).
xs-?I : xs D ([u] ?I (E u)) (?I F) <- xs D E F.
xs-?G : xs D ([u] ?G (E u)) (?G F) <- xs D E F.

xs-letcc : xs D ([u] letcc ([v] E v u)) (letcc ([v] F v))
        <- ({v : B * W'} xs D (E v) (F v)).

% throw to different cont
xs-throwmiss : xs D ([u] throw (E u) V) (throw F V)
            <- xs D E F.

% when reaching the throw, pass the term we're throwing
% to D instead.
xs-throwhit  : xs D ([u] throw (E u) u) (D B W' F)
            <- xs D E F.


%%% Translation from Sequent Calculus to Natural Deduction


%       G # D     then   G ; D |- M : *
seqnd :   #         -> ({a}{w} a @ w) -> type.  %name seqnd S.
truend :  true A W  -> A @ W -> type.           %name truend T t.
falsend : false A W -> A * W -> type.           %name falsend F f.
%mode seqnd +D -F.
%mode truend +D -F.
%mode falsend +D -F.

% judgmental
sn-contra : seqnd (contra AT AF) ([c : prop][w : world] throw A AC)
         <- truend AT A
         <- falsend AF AC.

% For each connective, the T side is easy -- it just corresponds to
% the elimination rule. The F side requires the "excluded substitution"
% theorem above, and is often quite tricky.

% falsehood
sn-_|_T : seqnd (_|_T FT) ([c][w] _|_E FT') <- truend FT FT'.

% hybrid
sn-atT : seqnd (atT ([aa] D aa) Tat)
         ([c][w] atE (get atmob Nat : _ @ W') ([a'] F a' c w))
     <- truend Tat Nat
     <- ({aa : true A W'}{a' : A @ W'}
           truend aa a' ->
         seqnd (D aa) ([c][w] F a' c w)).
sn-atF : seqnd (atF ([af'] D af') Fat) FF
     <- falsend Fat Nat
     <- ({af' : false A W'}{a*' : A * W'}
           falsend af' a*' ->
         seqnd (D af') ([c][w] F1 a*' c w))
     <- ({cc}{ww}
           xs ([c][w] [a'] (throw (get atmob (atI a')) Nat))
              ([a*'] F1 a*' cc ww)
              (FF cc ww)).

% conjunction
sn-&T : seqnd (&T ([a][bt] D a bt) T&) ([c][w] F (&E1 N&) (&E2 N&) c w)
     <- ({aa}{a}
```

```
           truend aa a ->
         {bb}{b}
           truend bb b ->
         seqnd (D aa bb) ([c][w] F a b c w))
     <- truend T& N&.

% x.\ldb y.\sthrow{\langle}x, y{\rangle} \sto up / ub \rdb N_2\, / ua \brdb N_1
sn-&F : seqnd (&F ([af] D1 af) ([bf] D2 bf) (F& : false (A & B) W))
             FF
     <- falsend F& N&
     <- ({af}{a*} falsend af a* ->
         seqnd (D1 af) ([c][w] F1 a* c w))
     <- ({bf}{b*} falsend bf b* ->
         seqnd (D2 bf) ([c][w] F2 b* c w))
     <- ({cc}{ww}{b : B @ W}
         xs ([c][w] [a] throw (&I a b) N&)
            ([a*] F1 a* cc ww)
            (F1' b cc ww))
     <- ({cc : prop}{ww}
         xs ([c][w] [b] F1' b c w)
            ([b*] F2 b* cc ww)
            (FF cc ww)).

% implication
% actually, implication is not just the elim rule, because
% classical implication is phrased differently.
sn-=>T : seqnd (=>T ([af : false A W] D1 af) ([bt] D2 bt)
                    (T=> : true (A => B) W))
              FF
      <- truend T=> N=>
      <- ({af}{a*} falsend af a* ->
          seqnd (D1 af) ([c][w] F1 a* c w))
      <- ({bt}{b}
            truend bt b ->
          seqnd (D2 bt) ([c][w] F2 b c w))
      <- ({cc}{ww}
            xs ([c][w] [a] (F2 (=>E N=> a) c w))
               ([a*] F1 a* cc ww)
               (FF cc ww)).

% letcc-free version
% u : A=>B |- throw \x:A . [[ c:B. throw \unused:A.c to u / ... ]](IH) to u
sn-=>F : seqnd (=>F ([aa : true A W][bf : false B W] D aa bf) F=>)
               ([c][w] throw (=>I [a] FZ a B W) N=> )
      <- falsend F=> N=>
      <- ({aa}{a}
            truend  aa a  ->
          {bf}{b*} falsend bf b* ->
          seqnd (D aa bf) ([c][w] F1 a b* c w))
      <- ({a : A @ W}
          {cc}{ww}
            xs ([c][w] [b] throw (=>I [a-unused : A @ W] b) N=>)
               ([b*] F1 a b* cc ww)
               (FZ a cc ww)).

% also include simpler letcc version
% u : A=>B |- throw (\x:A . letcc v : b* in (IH) end) to u
sn-=>F-letcc :
        seqnd (=>F ([aa : true A W][bf : false B W] D aa bf) F=>)
              ([c][w] throw (=>I [a] letcc [b*] F1 a b* B W) N=>)
          <- falsend F=> N=>
          <- ({aa}{a}
                truend  aa a  ->
              {bf}{b*} falsend bf b* ->
              seqnd (D aa bf) ([c][w] F1 a b* c w)).
```

```
% box
sn-!T : seqnd (!T ([aa' : true A W'] D aa') T!) ([c][w] F (!E (!G N!)) c w)
     <- ({aa'}{a'}
           truend aa' a' ->
         seqnd (D aa') ([c][w] F a' c w))
     <- truend T! N!.

% this is the only necessary letcc in the proof
% u : []A * w |- throw (box w'. letcc v:A*w' in (IH) end) to u
sn-!F : seqnd (!F ([w' : world][af : false A w'] D w' af) F!)
             ([c][w] (throw (!I [w'] letcc ([a*'] F w' a*' A w')) N!))
     <- falsend F! N!
     <- ({w' : world}{af' : false A w'}{a*' : A * w'}
           falsend af' a*' ->
         seqnd (D w' af') ([c][w] F w' a*' c w)).

% dia
% x : ?A@w |- let dia <y,w'> = get<w>x in IH end
sn-?T : seqnd (?T ([w'][aa] D w' aa) T?)
          ([c][w] ?E (?G N?) ([w'][a'] F w' a' c w))
     <- truend T? N?
     <- ({w'}{aa : true A w'}{a' : A @ w'}
           truend aa a' ->
         seqnd (D w' aa) ([c][w] F w' a' c w)).

sn-?F : seqnd (?F ([af'] D af') F?)
          FF
     <- falsend F? N?
     <- ({af' : false A W'}{a*' : A * W'}
           falsend af' a*' ->
         seqnd (D af') ([c][w] F1 a*' c w))
     <- ({cc}{ww}
           xs ([c][w] [a'] (throw (?G (?I a')) N?))
              ([a*'] F1 a*' cc ww)
              (FF cc ww)).

% for xs
%block blocku : some {A : prop} {W : world}
                block {u : A * W}.
%block blocka : some {A : prop} {W : world}
                block {u : A @ W}.

% for seqnd
%block blocknt : some {A:prop} {W:world}
                 block {aa:true A W} {a:A @ W} {t:truend aa a}.
%block blocknf : some {A:prop} {W:world}
                 block {af:false A W} {a:A * W} {t:falsend af a}.

%block blockp : block {a:prop}.

%worlds (blockw | blocka | blocknt | blocknf | blocku | blockp)
        (xs D E F) (seqnd D F) (truend D F) (falsend D F).

%total E (xs D E F).
%total (D T F) (seqnd D FF) (truend T N) (falsend F M).
```

A.5 Operational semantics and type safety of C5

The second signature requires the lemmas from the first, but otherwise these signatures stand alone.

A.5.1 Natural numbers

```
nat : type.               %name nat N n.

0 : nat.
s : nat -> nat.

< : nat -> nat -> type.   %infix none 5 <.
%name < LT lt.
lt0 : 0 < (s N).
lts : (s N1) < (s N2) <- N1 < N2.

plus : nat -> nat -> nat -> type.
p0 : plus N 0 N.
ps : plus N (s M) (s X) <- plus N M X.
%mode plus +N +M -X.

minus : nat -> nat -> nat -> type.
m0 : minus N 0 N.
ms : minus (s N) (s M) X <- minus N M X.
%mode minus +N +M -X.

pred : nat -> nat -> type.
preds : pred (s N) N.
%mode pred +N -M.

nat-eq : nat -> nat -> type.
nat-eqi : nat-eq N N.
%mode nat-eq +M +N.

minus-eq : minus X Y Z -> minus X Y W -> nat-eq Z W -> type.
%mode minus-eq +M1 +M2 -E.
minus-eq-m0 : minus-eq m0 m0 nat-eqi.
minus-eq-ms : minus-eq (ms M1 : minus (s X) (s Y) Z) (ms M2 : minus (s X) (s Y) W) E
           <- minus-eq M1 M2 E.

minus-self : {X : nat} minus X X 0 -> type.
%mode minus-self +X -M.
minus-self-0 : minus-self 0 m0.
minus-self-s : minus-self (s X) (ms M) <- minus-self X M.

minus-sfirst : minus X Y Z -> minus (s X) Y (s Z) -> type.
%mode minus-sfirst +M -M'.
minus-sfirst-0 : minus-sfirst m0 m0.
minus-sfirst-s : minus-sfirst (ms M) (ms M') <- minus-sfirst M M'.

succ-eq : nat-eq Q R -> nat-eq (s Q) (s R) -> type.
%mode succ-eq +M -N.
succ-eqi : succ-eq nat-eqi nat-eqi.

plus-zero : plus X 0 Y -> nat-eq X Y -> type.
%mode plus-zero +M -E.
plusz0 : plus-zero p0 nat-eqi.

plus-eq : plus X Y Z -> plus X Y W -> nat-eq Z W -> type.
%mode plus-eq +M1 +M2 -E.
plus-eq-p0 : plus-eq p0 p0 nat-eqi.
plus-eq-ps : plus-eq (ps M1 : plus X (s Y) (s Q))
                     (ps M2 : plus X (s Y) (s R)) EE
    <- plus-eq M1 M2 (E : nat-eq Q R)
    <- succ-eq E (EE : nat-eq (s Q) (s R)).

% if   a + (b + 1) = c   then  (a + 1) + b = c
plus-flip : plus X (s Y) Z -> plus (s X) Y Z -> type.
%mode plus-flip +P -PP.
plus-flip0 : plus-flip (ps p0 : plus X (s 0) (s X)) p0.
```

```
plus-flips : plus-flip (ps (ps P)) (ps PP) <- plus-flip (ps P) PP.

plus-lzero : {X : nat} plus 0 X X -> type.
%mode plus-lzero +X -P.
plus-lz0 : plus-lzero 0 p0.
plus-lzs : plus-lzero (s X) (ps P) <- plus-lzero X P.

plus-commutes : plus X Y Z -> plus Y X Z -> type.
%mode plus-commutes +P -PP.
plus-coms :  plus-flip (ps PP) PPP ->
             plus-commutes P PP ->
             plus-commutes (ps P) PPP.
plus-com0 :  plus-lzero X P -> plus-commutes p0 P.

plus-zero-eq : plus 0 X Y -> nat-eq X Y -> type.
plus-zero-0 : plus-zero-eq p0 nat-eqi.
plus-zero-s : plus-zero-eq (ps P) E' <- plus-zero-eq P E <- succ-eq E E'.
%mode plus-zero-eq +P -E.

eq-refl : nat-eq X Y -> nat-eq Y X -> type.
%mode eq-refl +E -E'.
eq-refli : eq-refl nat-eqi nat-eqi.

%worlds () (minus-eq _ _ _) (minus-self _ _) (minus-sfirst _ _)
  (succ-eq _ _) (plus-eq _ _ _) (plus-flip _ _) (plus-lzero _ _)
  (plus-commutes _ _) (plus-zero _ _) (plus-zero-eq _ _) (eq-refl _ _).

%total [M1 M2] (minus-eq M1 M2 E).
%total X (minus-self X _).
%total M (minus-sfirst M _).
%total E (succ-eq E EE).
%total M2 (plus-eq M1 M2 E).
%total P (plus-flip P PP).
%total X (plus-lzero X P).
%total P (plus-commutes P PP).
%total P (plus-zero P E).
%total P (plus-zero-eq P E).
%total E (eq-refl E _).
```

A.5.2 Operational semantics

```
%% Thanks: Jason Reed and Karl Crary

%%%%
%%%%   Types.
%%%%

world : type.               %name world W w.
typ : type.                 %name typ A a.

&   : typ -> typ -> typ.    %infix none 9 &.
=>  : typ -> typ -> typ.    %infix right 8 =>.
!   : typ -> typ.           %prefix 9 !.
?   : typ -> typ.           %prefix 9 ?.
at  : typ -> world -> typ.  %infix none 4 at.
_|_ : typ.

mobile : typ -> type.       %name mobile M m.
!mob  : mobile (! A).
?mob  : mobile (? A).
atmob : mobile (A at W).
&mob  : mobile A -> mobile B -> mobile (A & B).


%%%%
```

```
%%%%   Syntax of expressions.
%%%%

% now we have constant worlds
wconst : nat -> world.

% expressions and continuation expressions
exp  : type.   %name exp M m.
cexp : type.   %name cexp C c.

% structural
get : world -> exp -> exp.

% products
pair : exp -> exp -> exp.
fst : exp -> exp.
snd : exp -> exp.

% implication
app : exp -> exp -> exp.
lam : (exp -> exp) -> exp.

% box
box : (world -> exp) -> exp.
unbox : exp -> exp.

% dia
here : exp -> exp.
letdia : exp -> (world -> exp -> exp) -> exp.

% falsehood
abort : exp -> exp.

% classical stuff
throw : exp -> cexp -> exp.
letcc : (cexp -> exp) -> exp.

% hybrid
hold : exp -> exp.
leta : exp -> (exp -> exp) -> exp.

% new: runtime artifacts
lab : nat -> exp.
addr : world -> nat -> exp.
caddr : world -> nat -> cexp.
held : exp -> exp.

% proofs of value-ness
value : exp -> type.

valbox  : value (box _).
valaddr : value (addr _ _).
valpair : value E1 -> value E2 -> value (pair E1 E2).
vallam  : value (lam _).
valheld : value V -> value (held V).

%%%%
%%%%   Syntax of continuations.
%%%%

cont : type.      %name cont K k.
finish : cont.
fabort : cont.
ffst : cont -> cont.
fsnd : cont -> cont.
funbox : cont -> cont.
```

```
freturn : nat -> nat -> cont.
fletdia : cont -> (world -> exp -> exp) -> cont.
fpair1 : cont -> exp -> cont. % k |> ([], M)
fpair2 : cont -> {V : exp} value V -> cont. % k |> (v, [])
fhere : cont -> cont.
fapp1 : cont -> exp -> cont. % k |> ([] M)
fapp2 : cont -> {V : exp} value V -> cont. % k |> (v [])
fhold : cont -> cont.
fleta : cont -> (exp -> exp) -> cont.

%%%%
%%%%   Networks.
%%%%

% pass in size of network (number of worlds)
network : nat -> type.    %name network NN net.

% world configurations have tables of values
vlist : nat -> type.
vnil : vlist 0.
v// : {V : exp} value V -> vlist N -> vlist (s N).

% and tables of continuations
clist : nat -> type.
cnil : clist 0.
c// : cont -> clist N -> clist (s N).

% lists of world data
wdlist : nat -> type.      %name wdlist W w.
wdnil : wdlist 0.
%      cont table  value table
wd// :   clist X -> vlist Y -> wdlist N -> wdlist (s N).

% mode of evaluation
mode : type.
eval : exp -> mode.
ret  : {V : exp} value V -> mode.

net : % configuration of worlds
      wdlist N ->
      % current world
      {cur : nat}
      % current continuation
      cont ->
      % current mode (eval exp  -or-  ret v)
      mode ->
      network N.

%%%%
%%%%   Store (configuration) typing.
%%%%

% tables mapping numbers to types
% this is used for both continuation table types
% and value table types
typetable : nat -> type.
ttnil : typetable 0.
tt// : typ -> typetable N -> typetable (s N).

% lists of world types
wslist : nat -> type.     %name wslist S s.
wsnil : wslist 0.
% continuation table, value table, tail
ws// : typetable X -> typetable Y -> wslist N -> wslist (s N).

%%%%
```

```
%%%%   Typing labels
%%%%

% typing for labels is tricky because we have to look in
% the store type.

% assert that b(n) = A. Note that labels count up from
% the end of the list, so the user of this relation must
% first compute the actual offset
tt_sub : typetable X -> nat -> typ -> type.

tts_found : tt_sub (tt// A _) 0 A.
tts_next  : tt_sub TL N A ->
            tt_sub (tt// _ TL) (s N) A.


% assert that S |- lab : A @ W.
%             store      lab     A     world
labtype_is : wslist N -> nat -> typ -> nat -> type.

% offset of this entry is (size - 1) - label
lti_found : minus SIZE-1 L OFF ->
             tt_sub VT OFF A ->
             labtype_is (ws// _ (VT : typetable (s SIZE-1)) _) L A 0.

lti_next  :  labtype_is TL L A W ->
             labtype_is (ws// _ _ TL) L A (s W).

% same, looks in continuation table
clabtype_is : wslist N -> nat -> typ -> nat -> type.
%name clabtype_is CLTI clti.

clti_found : minus SIZE-1 L OFF ->
             tt_sub CT OFF A ->
             clabtype_is (ws// (CT : typetable (s SIZE-1)) _ _) L A 0.

clti_next  : clabtype_is TL L A W ->
             clabtype_is (ws// _ _ TL) L A (s W).

%%%%
%%%%   Typing of worlds.
%%%%

% essentially, just that the world exists
welltyped_world : wslist X -> world -> type.
%name welltyped_world WTW wtw.

wtw_const : {S : wslist X} J < X ->
            welltyped_world S (wconst J).

%%%%
%%%%   Typing of continuation expressions.
%%%%

welltyped_cexp : wslist X -> cexp -> typ -> world -> type.

% continuation address constants
wtce_caddr : clabtype_is S L A J ->
             welltyped_cexp S (caddr (wconst J) L) A (wconst J).

%%%%
%%%%   Typing of expressions.
%%%%

welltyped_exp  : wslist X -> exp -> typ -> world -> type.
wte_pair :  welltyped_exp S M A W ->
```

```
            welltyped_exp S N B W ->
            welltyped_exp S (pair M N) (A & B) W.

wte_fst :   welltyped_exp S M (A & B) W ->
            welltyped_exp S (fst M) A W.
wte_snd :   welltyped_exp S M (A & B) W ->
            welltyped_exp S (snd M) B W.

wte_hold :  welltyped_exp S M A W ->
            welltyped_exp S (hold M) (A at W) W.

% the value allows the contents to be remote
wte_held :  welltyped_exp S M A W' ->
            value M ->
            welltyped_exp S (held M) (A at W') W.

wte_leta : welltyped_exp S M (A at W') W ->
            ({x : exp} {wt : {X : nat}{S : wslist X}
               welltyped_exp S x A W'}
             welltyped_exp S (N x) C W) ->
            welltyped_exp S (leta M N) C W.

wte_here :  welltyped_exp S M A W ->
            welltyped_exp S (here M) (? A) W.

wte_app :   welltyped_exp S M (A => B) W ->
            welltyped_exp S N A W ->
            welltyped_exp S (app M N) B W.

wte_lam :   ({x : exp} {w : {X : nat}{S : wslist X} welltyped_exp S x A W}
             welltyped_exp S (M x) B W) ->
            welltyped_exp S (lam ([x] M x)) (A => B) W.

wte_letdia : welltyped_exp S M (? A) W ->
            ({w : world} {wt1 : {X : nat}{S : wslist X}
                                welltyped_world S w }
               {x : exp} {wt2 : {X : nat}{S : wslist X}
                                welltyped_exp S x A w}
             welltyped_exp S (N w x) C W) ->
            welltyped_exp S (letdia M N) C W.

wte_box :   ({w : world} {wt : {X : nat}{S : wslist X}
                               welltyped_world S w }
               welltyped_exp S (M w) A w) ->
            welltyped_exp S (box ([w] M w)) (! A) W.

wte_unbox : welltyped_exp S M (! A) W ->
            welltyped_exp S (unbox M) A W.

wte_letcc : ({u : cexp} {wt : {X : nat}{S : wslist X} welltyped_cexp S u A W}
               welltyped_exp S (M u) A W) ->
            welltyped_exp S (letcc M) A W.

wte_throw : welltyped_cexp S U A W ->
            welltyped_exp S M A W ->
            welltyped_exp S (throw M U) C W'.

wte_get : welltyped_world S W' ->
          welltyped_exp S M A W' ->
          mobile A ->
          welltyped_exp S (get W' M) A W.

wte_abort : welltyped_exp S M _|_ W ->
            welltyped_exp S (abort M) C W.

wte_lab : labtype_is S L A W ->
```

```
          welltyped_exp (S : wslist N) (lab L) A (wconst W).

% addrs are like labels
wte_addr : labtype_is S L A J ->
           welltyped_exp S (addr (wconst J) L) (? A) (wconst _).


%%%%
%%%%   Typing of continuations.
%%%%

welltyped_cont : wslist N -> cont -> typ -> world -> type.
%name welltyped_cont WTC wtc.
wtc_finish : welltyped_cont _ finish _ _.
wtc_fabort : welltyped_cont S fabort _|_ W.
wtc_freturn : clabtype_is S L A J ->
              mobile A ->
              welltyped_cont S (freturn J L) A W.

wtc_fst : welltyped_cont S K A W ->
          welltyped_cont S (ffst K) (A & _) W.

wtc_snd : welltyped_cont S K B W ->
          welltyped_cont S (fsnd K) (_ & B) W.

wtc_pair1 : welltyped_exp  S M B W ->
            welltyped_cont S K (A & B) W ->
            welltyped_cont S (fpair1 K M) A W.

wtc_pair2 : welltyped_exp  S V A W ->
            welltyped_cont S K (A & B) W ->
            welltyped_cont S (fpair2 K V VP) B W.

wtc_here : welltyped_cont S K (? A) W ->
           welltyped_cont S (fhere K) A W.

wtc_hold : welltyped_cont S K (A at W) W ->
           welltyped_cont S (fhold K) A W.

wtc_app1 : welltyped_exp S M A W ->
           welltyped_cont S K B W ->
           welltyped_cont S (fapp1 K M) (A => B) W.

wtc_app2 : welltyped_exp S V (A => B) W ->
           welltyped_cont S K B W ->
           welltyped_cont S (fapp2 K V VP) A W.

wtc_unbox : welltyped_cont S K A W ->
            welltyped_cont S (funbox K) (! A) W.

wtc_letdia : welltyped_cont S K C W ->
             ({w : world}
                {wtw : {X : nat}{SS : wslist X}
                        welltyped_world SS w}
              {x : exp}{wt : {X : nat}{SS : wslist X}
                             welltyped_exp SS x A w}
                welltyped_exp S (N w x) C W) ->
             welltyped_cont S (fletdia K N) (? A) W.

wtc_leta :   welltyped_cont S K C W ->
             ({x : exp}{wt : {X : nat}{SS : wslist X}
                             welltyped_exp SS x A W'}
                welltyped_exp S (N x) C W) ->
             welltyped_cont S (fleta K N) (A at W') W.

% typing of expressions and continuations both respect equality on
```

```
% worlds

wte-resp : {J : nat} { J' : nat } nat-eq J J' ->
           welltyped_exp S M A (wconst J) ->
           welltyped_exp S M A (wconst J') -> type.
%mode wte-resp +J +JJ' +N +W -W'.
wte-respi : wte-resp _ _ nat-eqi D D.

wtc-resp : {J : nat} { J' : nat } nat-eq J J' ->
           welltyped_cont S K A (wconst J) ->
           welltyped_cont S K A (wconst J') -> type.
%mode wtc-resp +J +JJ' +N +W -W'.
wtc-respi : wtc-resp _ _ nat-eqi D D.

%%%%
%%%%   Typing of network bits.
%%%%

welltyped_mode : wslist N -> mode -> typ -> world -> type.
wt_eval : welltyped_exp S M A W ->
          welltyped_mode S (eval M) A W.
wt_ret  : welltyped_exp S M A W ->
          welltyped_mode S (ret M _) A W.

% ensure that a continuation table has the given type (at the supplied world)
% under the given store type.
welltyped_conts : wslist N -> typetable M -> clist M -> world -> type.
wtcs_empty : welltyped_conts S ttnil cnil W.
wtcs_one : welltyped_conts S TTL CTL W ->
           welltyped_cont S K A W ->
           welltyped_conts S (tt// A TTL) (c// K CTL) W.

% same, but data table
welltyped_datas : wslist N -> typetable M -> vlist M -> world -> type.
wtds_empty : welltyped_datas S ttnil vnil W.
wtds_one : welltyped_datas S TTL VTL W ->
           welltyped_exp S V A W ->
           welltyped_datas S (tt// A TTL) (v// V _ VTL) W.

% the below requires a nested induction
welltyped_worlds' : % size of rest
                    {N : nat}
                    % current index
                    nat ->
                    % full store and rest of store
                    wslist M -> wslist N ->
                    % world data
                    wdlist N -> type.

wtws'_empty : welltyped_worlds' 0 _ _ wsnil wdnil.
wtws'_one : welltyped_worlds' N (s IDX) S STL WTL ->
            welltyped_conts   S CT CD (wconst IDX) ->
            welltyped_datas   S BT BD (wconst IDX) ->
            welltyped_worlds' (s N) IDX S
            (ws// CT BT STL) (wd// CD BD WTL).

% ensure that the world configuration has the store type
% indicated (in greater store type M)
welltyped_worlds : wslist N -> wdlist N -> type.
wtws : welltyped_worlds' N 0 S S L ->
       welltyped_worlds (S : wslist N) L.

% sometimes we aggregate wtws in reverse
welltyped_worldsr : {M : nat} wslist X -> wslist M -> wdlist M -> type.
wtwsr_none : welltyped_worldsr 0 _ wsnil wdnil.
wtwsr_one  : welltyped_worldsr N-1 S STL WTL ->
```

```
            welltyped_conts S CT CD (wconst N-1) ->
            welltyped_datas S BT BD (wconst N-1) ->
            welltyped_worldsr (s N-1) S (ws// CT BT STL) (wd// CD BD WTL).

%%%%
%%%%   Typing of networks.
%%%%

% type of network (store typing)
wellformed : wslist N -> network N -> type.
wf_net : % current world must be in bounds
         J < N ->
         % must be a mediating type for mode/cont
         {A : typ}
         welltyped_mode S M A (wconst J) ->
         welltyped_cont S K A (wconst J) ->
         welltyped_worlds S W ->
         wellformed (S : wslist N) (net W J K M : network N).

% extendst T1 T2   iff  T1 is a suffix of T2
% (specialized to the cases where the difference is either 0 or 1)
extendst : typetable X -> typetable Y -> type.

extendst-same : extendst TT TT.
extendst-cons : extendst TT (tt// _ TT).

% extends S S' if S' has strictly more information,
% but they agree on the domain of S.
extends : wslist N -> wslist N -> type.
%name extends EX ex.

ex-none : extends wsnil wsnil.
ex-one  : extends S S' ->
          extendst CT CT' ->
          extendst BT BT' ->
          extends (ws// CT BT S) (ws// CT' BT' S').

% lemma: a store type extends itself. For most cases of preservation
% we don't change the store, so this gets used all over the place
ex-id : {S : wslist N} extends S S -> type.
%mode ex-id +S -E.

ex-id-none : ex-id wsnil ex-none.
ex-id-one : ex-id S E ->
            ex-id (ws// CT BT S) (ex-one E extendst-same extendst-same).

%worlds () (ex-id S E).
%total S (ex-id S E).

%%%%
%%%%   Lookup of labels.
%%%%

% lookin B O V: the table B contains value V at offset O
lookinv : vlist N -> nat -> {V : exp} value V -> type.   %name lookinv LIV liv.
lookinv_next : lookinv (v// _ _ VTL) (s OFF) V VP
            <- lookinv VTL OFF V VP.

lookinv_found : lookinv (v// V VP _) 0 V VP.

lookinv-resp : nat-eq L L' -> lookinv VL L V VP -> lookinv VL L' V VP -> type.
%mode lookinv-resp +B +C -D.
lookinv-respi : lookinv-resp nat-eqi D D.

lookinc : clist N -> nat -> cont -> type.                %name lookinc LIC lic.
lookinc_next : lookinc (c// _ VTL) (s OFF) K
```

```
            <- lookinc VTL OFF K.

lookinc_found : lookinc (c// K _) 0 K.

lookinc-resp : nat-eq L L' -> lookinc VL L K -> lookinc VL L' K -> type.
%mode lookinc-resp +B +C -D.
lookinc-respi : lookinc-resp nat-eqi D D.


% lookupv W J L V
% ensure that in config W at world constant J, label L maps to some value V.
lookupv : wdlist N -> nat -> nat -> {V : exp} value V -> type.
lookupv_next : lookupv (wd// _ BD BTL) (s X) L V VP
            <- lookupv BTL X L V VP.

lookupv_found :
     minus SIZE-1 L OFF ->
     lookinv BD OFF V VP ->
     lookupv (wd// _ (BD : vlist (s SIZE-1)) _) 0 L V VP.

% add_value W J V VAL W' L
% adding value V to the world J within W results in worlds
% W' and new label L.
add_value : wdlist N -> nat -> {V : exp} value V -> wdlist N -> nat -> type.
%mode add_value +W +J +V +VP -W' -L.
addv_next : add_value (wd// CD BD BTL) (s X) V VP (wd// CD BD BTL') L
         <- add_value BTL X V VP BTL' L.

% adds to the head of the list, which gets index n (the current length)
addv_found :
    {BD : vlist N}
    add_value (wd// CD BD BTL) 0 V VP (wd// CD (v// V VP BD) BTL) N.

% same, but for continuations
lookupc : wdlist N -> nat -> nat -> cont -> type.
%name lookupc LUC luc.
lookupc_next : lookupc (wd// _ _ BTL) (s X) L K
            <- lookupc BTL X L K.

lookupc_found :
     minus SIZE-1 L OFF ->
     lookinc CD OFF K ->
     lookupc (wd// (CD : clist (s SIZE-1)) _ _) 0 L K.

add_cont : wdlist N -> nat -> cont -> wdlist N -> nat -> type.
%mode add_cont +W +J +K -W' -L.

addc_next : add_cont (wd// CD BD BTL) (s X) K (wd// CD BD BTL') L
         <- add_cont BTL X K BTL' L.

addc_found : {CD : clist N}
             add_cont (wd// CD BD BTL) 0 K (wd// (c// K CD) BD BTL) N.

%%%%
%%%%   Stepping relation.
%%%%

% stepping relation between networks of the same size.
|-> : network N -> network N -> type.        %infix none 0 |->.

% little trick: terminal network steps to itself
% this makes the statement of progress easier.
step_done : net W J finish (ret V VP) |-> net W J finish (ret V VP).

step_lab : lookupv W J L V VP ->
           net W J K (eval (lab L)) |-> net W J K (ret V VP).
```

```
step_pfst : net W J K (eval (fst M)) |-> net W J (ffst K) (eval M).
step_psnd : net W J K (eval (snd M)) |-> net W J (fsnd K) (eval M).
step_rfst : net W J (ffst K) (ret (pair V _) (valpair VP _)) |-> net W J K (ret V VP).
step_rsnd : net W J (fsnd K) (ret (pair _ V) (valpair _ VP)) |-> net W J K (ret V VP).
step_ppair : net W J K (eval (pair M N)) |-> net W J (fpair1 K N) (eval M).
step_fpair : net W J (fpair1 K N) (ret V VP) |-> net W J (fpair2 K V VP) (eval N).
step_rpair : net W J (fpair2 K V VP) (ret VV VVP) |->
             net W J K (ret (pair V VV) (valpair VP VVP)).
step_phere : net W J K (eval (here M)) |-> net W J (fhere K) (eval M).
step_rhere : add_value W J V VP W' L ->
        net W J (fhere K) (ret V VP) |-> net W' J K (ret (addr (wconst J) L) valaddr).
step_phold : net W J K (eval (hold M)) |-> net W J (fhold K) (eval M).
step_rhold : net W J (fhold K) (ret V VP) |-> net W J K (ret (held V) (valheld VP)).
step_papp : net W J K (eval (app M N)) |-> net W J (fapp1 K N) (eval M).
step_fapp : net W J (fapp1 K N) (ret V VP) |-> net W J (fapp2 K V VP) (eval N).
step_rapp : net W J (fapp2 K (lam F) _) (ret V _) |-> net W J K (eval (F V)).

% we need to flip from eval to ret when encountering certain expressions
step_taddr : net W J K (eval (addr X L)) |-> net W J K (ret (addr X L) valaddr).
step_theld : net W J K (eval (held V)) |-> net W J K (ret (held V) (valheld VP)).
step_tbox : net W J K (eval (box M)) |-> net W J K (ret (box M) valbox).
step_tlam : net W J K (eval (lam B)) |-> net W J K (ret (lam B) vallam).
step_punbox : net W J K (eval (unbox M)) |-> net W J (funbox K) (eval M).
step_runbox : net W J (funbox K) (ret (box F) valbox) |->
              net W J K (eval (F (wconst J))).
step_pletdia : net W J K (eval (letdia M N)) |-> net W J (fletdia K N) (eval M).
step_rletdia : net W J (fletdia K N) (ret (addr WW LL) valaddr) |->
               net W J K (eval (N WW (lab LL))).
step_pleta : net W J K (eval (leta M N)) |-> net W J (fleta K N) (eval M).
step_rleta : net W J (fleta K N) (ret (held V) (valheld _)) |->
             net W J K (eval (N V)).
step_rletcc : add_cont W J K W' L ->
              net W J K (eval (letcc M)) |->
              net W' J K (eval (M (caddr (wconst J) L))).
step_rreturn : lookupc W JNEW L K ->
               net W JOLD (freturn JNEW L) (ret M V) |->
               net W JNEW K (ret M V).
step_throw : lookupc W JNEW L K ->
             net W JOLD _ (eval (throw M (caddr (wconst JNEW) L))) |->
             net W JNEW K (eval M).
step_get : add_cont W JOLD K W' L ->
           net W JOLD K (eval (get (wconst JNEW) M)) |->
           net W' JNEW (freturn JOLD L) (eval M).
step_abort : net W J _ (eval (abort M)) |-> net W J fabort (eval M).

%%%%
%%%%   Lemmas.
%%%%

% lemmas about weakening.

weaken-lti : {S  : wslist N} {S' : wslist N} extends S S' ->
             labtype_is S L A W -> labtype_is S' L A W -> type.
%mode weaken-lti +S +S' +E +LTI -LTI'.

wlti-next :
    weaken-lti SL SL' E LTI LTI' ->
    weaken-lti (ws// CT BT SL) (ws// CT' BT' SL') (ex-one E _ _)
             (lti_next LTI) (lti_next LTI').

wlti-foundsame :
    weaken-lti _ _ (ex-one _ _ extendst-same)
             (lti_found MINUS TTS) (lti_found MINUS TTS).

wlti-foundcons :
```

```
    minus-sfirst MINUS MINUS' ->
    weaken-lti (ws// _ (BT : typetable (s SIZE-1)) _)
               (ws// _ (tt// A BT) _) (ex-one _ _ extendst-cons)
             (lti_found MINUS TTS) (lti_found MINUS' (tts_next TTS)).

%% same as above, for clabtypes

weaken-clti : {S  : wslist N} {S' : wslist N} extends S S' ->
              clabtype_is S L A W -> clabtype_is S' L A W -> type.
%mode weaken-clti +S +S' +E +CLTI -CLTI'.

wclti-next :
    weaken-clti SL SL' E CLTI CLTI' ->
    weaken-clti (ws// CT BT SL) (ws// CT' BT' SL') (ex-one E _ _)
             (clti_next CLTI) (clti_next CLTI').

wclti-foundsame :
    weaken-clti _ _ (ex-one _ extendst-same _)
             (clti_found MINUS TTS) (clti_found MINUS TTS).

wclti-foundcons :
    minus-sfirst MINUS MINUS' ->
    weaken-clti (ws// (CT : typetable (s SIZE-1)) _ _)
               (ws// (tt// A CT) _ _) (ex-one _ extendst-cons _)
             (clti_found MINUS TTS) (clti_found MINUS' (tts_next TTS)).

%% worlds
weaken-wtw : extends S S' ->
             welltyped_world S W ->
             welltyped_world S' W -> type.
%mode weaken-wtw +E +WT -WT'.

wwtw-const : weaken-wtw EX (wtw_const S LT) (wtw_const S' LT).

%% continuation expressions
weaken-wtce : extends S S' ->
              welltyped_cexp S U A W ->
              welltyped_cexp S' U A W -> type.
%mode weaken-wtce +E +WT -WT'.

wwtce-caddr : weaken-clti _ _ EX CLTI CLTI' ->
              weaken-wtce EX (wtce_caddr CLTI) (wtce_caddr CLTI').

%% expressions
weaken-wte : extends S S' ->
             welltyped_exp S M A W ->
             welltyped_exp S' M A W -> type.
%mode weaken-wte +E +WT -WT'.

wwte-throw :
     weaken-wtce EX CW CW' ->
     weaken-wte EX W W' ->
     weaken-wte EX (wte_throw CW W) (wte_throw CW' W').

wwte-unbox : weaken-wte EX W W' ->
             weaken-wte EX (wte_unbox W) (wte_unbox W').

wwte-lab : weaken-lti _ _ EX LTI LTI' ->
           weaken-wte EX (wte_lab LTI) (wte_lab LTI').

wwte-addr : weaken-lti _ _ EX LTI LTI' ->
            weaken-wte EX (wte_addr LTI) (wte_addr LTI').

wwte-pair : weaken-wte EX W1 W1' ->
            weaken-wte EX W2 W2' ->
            weaken-wte EX (wte_pair W1 W2) (wte_pair W1' W2').
```

```
wwte-app : weaken-wte EX W1 W1' ->
           weaken-wte EX W2 W2' ->
           weaken-wte EX (wte_app W1 W2) (wte_app W1' W2').

wwte-box :
     ({w : world} {wt : {X : nat} {S : wslist X}
                        welltyped_world S w}
                  {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
                         {EX : extends s s'}
                         weaken-wtw EX (wt Y s) (wt Y s')}
        weaken-wte EX (WT w wt) (WT' w wt)) ->
     weaken-wte EX (wte_box WT) (wte_box WT').

% we have to do a little trick here. it seems there's no way
% to prove directly what we want -- to hypothesize a well-typing
% for x and assume that there exists some weakening of it.
% instead, we assume a ridiculous thing: that there is a well-typing
% for every possible store. later we will substitute this out with
% another theorem.
wwte-lam :
    ({x : exp}
       {w : {X : nat} {S : wslist X} welltyped_exp S x A W}
       {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
              {EX : extends s s'}
              weaken-wte EX (w Y s) (w Y s')}
        weaken-wte EX (W1 x w) (W1' x w)) ->
     weaken-wte EX (wte_lam ([x] [wt] W1 x wt)) (wte_lam ([x] [wt] W1' x wt)).

% ditto for this binder
wwte-letdia :
     weaken-wte EX M M' ->
     ({w : world}
        {wtw : {X : nat}{S : wslist X}
               welltyped_world S w }
        {thm0 : {Y : nat} {s : wslist Y} {s' : wslist Y}
                {EX : extends s s'}
                weaken-wtw EX (wtw Y s) (wtw Y s')}
       {x : exp}
       {wt : {X : nat} {S : wslist X} welltyped_exp S x A w}
       {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
              {EX : extends s s'}
              weaken-wte EX (wt Y s) (wt Y s')}
        weaken-wte EX (N w wtw x wt) (N' w wtw x wt)) ->
     weaken-wte EX (wte_letdia M ([w][x][wt] N w x wt))
                   (wte_letdia M' ([w][x][wt] N' w x wt)).

wwte-leta :
     weaken-wte EX M M' ->
     ( {x : exp}
       {wt : {X : nat} {S : wslist X} welltyped_exp S x A W'}
       {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
              {EX : extends s s'}
              weaken-wte EX (wt Y s) (wt Y s')}
        weaken-wte EX (N x wt) (N' x wt)) ->
     weaken-wte EX (wte_leta M  ([x][wt] N x wt))
                   (wte_leta M' ([x][wt] N' x wt)).

wwte-letcc :
     ({u : cexp}
      {w : {X : nat} {S : wslist X} welltyped_cexp S u A W}
      {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
             {EX : extends s s'}
             weaken-wtce EX (w Y s) (w Y s')}
        weaken-wte EX (M u w) (M' u w)) ->
     weaken-wte EX (wte_letcc ([u][wt] M u wt))
```

```
                  (wte_letcc ([u][wt] M' u wt)).

wwte-fst : weaken-wte EX W W' ->
           weaken-wte EX (wte_fst W) (wte_fst W').
wwte-snd : weaken-wte EX W W' ->
           weaken-wte EX (wte_snd W) (wte_snd W').
wwte-here : weaken-wte EX W W' ->
            weaken-wte EX (wte_here W) (wte_here W').
wwte-hold : weaken-wte EX W W' ->
            weaken-wte EX (wte_hold W) (wte_hold W').
wwte-held : weaken-wte EX W W' ->
            weaken-wte EX (wte_held W VAL) (wte_held W' VAL).
wwte-get : weaken-wtw EX WO WO' ->
           weaken-wte EX W W' ->
           weaken-wte EX (wte_get WO W MOB) (wte_get WO' W' MOB).
wwte-abort : weaken-wte EX WF WF' ->
             weaken-wte EX (wte_abort WF) (wte_abort WF').
weaken-wtc : extends S S' ->
             welltyped_cont S K A J ->
             welltyped_cont S' K A J -> type.
%mode weaken-wtc +E +W -W'.

wwtc-finish : weaken-wtc _ wtc_finish wtc_finish.
wwtc-fabort : weaken-wtc _ wtc_fabort wtc_fabort.
wwtc-freturn : weaken-clti _ _ E CLTI CLTI' ->
               weaken-wtc E (wtc_freturn CLTI MOB) (wtc_freturn CLTI' MOB).
wwtc-fst : weaken-wtc E K K' ->
           weaken-wtc E (wtc_fst K) (wtc_fst K').
wwtc-snd : weaken-wtc E K K' ->
           weaken-wtc E (wtc_snd K) (wtc_snd K').
wwtc-pair1 : weaken-wtc E K K' ->
             weaken-wte E M M' ->
             weaken-wtc E (wtc_pair1 M K) (wtc_pair1 M' K').
wwtc-pair2 : weaken-wtc E K K' ->
             weaken-wte E M M' ->
             weaken-wtc E (wtc_pair2 M K) (wtc_pair2 M' K').
wwtc-unbox : weaken-wtc E K K' ->
             weaken-wtc E (wtc_unbox K) (wtc_unbox K').
wwtc-here : weaken-wtc E K K' ->
            weaken-wtc E (wtc_here K) (wtc_here K').
wwtc-hold : weaken-wtc E K K' ->
            weaken-wtc E (wtc_hold K) (wtc_hold K').
wwtc-app1 : weaken-wtc E K K' ->
            weaken-wte E M M' ->
            weaken-wtc E (wtc_app1 M K) (wtc_app1 M' K').
wwtc-app2 : weaken-wtc E K K' ->
            weaken-wte E M M' ->
            weaken-wtc E (wtc_app2 M K) (wtc_app2 M' K').

wwtc-letdia : weaken-wtc E K K' ->
              ({w : world}
                {wtw : {X : nat}{S : wslist X}
                       welltyped_world S w }
                {thm0 : {Y : nat} {s : wslist Y} {s' : wslist Y}
                        {EX : extends s s'}
                        weaken-wtw EX (wtw Y s) (wtw Y s')}
                   {x : exp}{wt : {X : nat}{SS : wslist X}
                                  welltyped_exp SS x A w}
                   {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
                           {EX : extends s s'}
                           weaken-wte EX (wt Y s) (wt Y s')}
                    weaken-wte E (M w wtw x wt) (M' w wtw x wt)) ->
              weaken-wtc E (wtc_letdia K M) (wtc_letdia K' M').

wwtc-leta : weaken-wtc E K K' ->
            ( {x : exp}{wt : {X : nat}{SS : wslist X}
```

```
                          welltyped_exp SS x A W'}
              {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
                     {EX : extends s s'}
                     weaken-wte EX (wt Y s) (wt Y s')}
               weaken-wte E (M x wt) (M' x wt)) ->
            weaken-wtc E (wtc_leta K M) (wtc_leta K' M').

weaken-wtds : extends S S' ->
              welltyped_datas S  TT VT W ->
              welltyped_datas S' TT VT W -> type.
%mode weaken-wtds +E +WD -WD'.

wwtds-empty : weaken-wtds _ wtds_empty wtds_empty.
wwtds-one : weaken-wtds E DS DS' ->
            weaken-wte E WE WE' ->
            weaken-wtds E (wtds_one DS WE) (wtds_one DS' WE').

weaken-wtcs : extends S S' ->
              welltyped_conts S  TT VT W ->
              welltyped_conts S' TT VT W -> type.
%mode weaken-wtcs +E +WD -WD'.

wwtcs-empty : weaken-wtcs _ wtcs_empty wtcs_empty.
wwtcs-one : weaken-wtcs E CS CS' ->
            weaken-wtc E K K' ->
            weaken-wtcs E (wtcs_one CS K) (wtcs_one CS' K').

weaken-wtws : extends S S' ->
              welltyped_worlds' N IDX S  SL WL ->
              welltyped_worlds' N IDX S' SL WL -> type.
%mode weaken-wtws +E +WD -WD'.

wwtws-empty : weaken-wtws _ wtws'_empty wtws'_empty.
wwtws-one   : weaken-wtws S WT WT' ->
              weaken-wtds S WD WD' ->
              weaken-wtcs S WC WC' ->
              weaken-wtws S (wtws'_one WT WC WD) (wtws'_one WT' WC' WD').

%block bm :
  some {A : typ} {J : nat}
  block {m:exp}
        {wt : {X : nat} {SS : wslist X} welltyped_exp SS m A (wconst J)}
        {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
           {ex : extends s s'} weaken-wte ex (wt Y s) (wt Y s')}.

%block bx :
  some {A : typ} {W : world}
  block {x : exp}
        {w : {X : nat} {S : wslist X} welltyped_exp S x A W}
        {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
           {EX : extends s s'} weaken-wte EX (w Y s) (w Y s')}.

%block bu :
  some {A : typ} {W : world}
  block {u : cexp}
        {w : {X : nat} {S : wslist X} welltyped_cexp S u A W}
        {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
           {EX : extends s s'} weaken-wtce EX (w Y s) (w Y s')}.

%block bww :
  block {w : world}
        {wtw : {X : nat}{S : wslist X} welltyped_world S w }
        {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
           {EX : extends s s'} weaken-wtw EX (wtw Y s) (wtw Y s')}.
```

```
% substitution for magic world hypotheses in expression typing
% derivations. Sticks in the constant world J; all we need to
% know is that it's in the domain of the store type (J < SN).

wte-wsubst : {J : nat} J < SN ->
             ({wt : {X : nat} {SS : wslist X} welltyped_world SS (wconst J)}
                welltyped_exp S M B WB) ->
             welltyped_exp (S : wslist SN) M B WB -> type.
%mode wte-wsubst +D1 +D2 +D2 -D3.

wtews-closed : wte-wsubst J LT ([wt] M) M.

wtews-rpc : wte-wsubst J LT WT WT' ->
            wte-wsubst J LT ([wt] wte_abort (WT wt)) (wte_abort WT').

% uses var
wtews-getv : wte-wsubst J LT ([wt] WT wt) WT' ->
             wte-wsubst J LT ([wt : {X : nat} {SS : wslist X}
                                    welltyped_world SS (wconst J)]
                                wte_get (wt _ S) (WT wt) MOB)
                              (wte_get (wtw_const S LT) WT' MOB).

% doesn't use var
wtews-get : wte-wsubst J LT ([wt] WT wt) WT' ->
            wte-wsubst J LT ([wt] wte_get WW (WT wt) MOB) (wte_get WW WT' MOB).

wtews-throw : wte-wsubst J LT ([wt] WT wt) WT' ->
              wte-wsubst J LT ([wt] wte_throw WC (WT wt)) (wte_throw WC WT').

wtews-unbox :
            wte-wsubst J LT ([wt] WT wt) WT' ->
            wte-wsubst J LT ([wt] wte_unbox (WT wt)) (wte_unbox WT').

wtews-app : wte-wsubst J LT ([w] WF w) WF' ->
            wte-wsubst J LT ([w] WB w) WB' ->
            wte-wsubst J LT ([w : {X : nat} {SS : wslist X}
                                welltyped_world SS (wconst J)]
                              wte_app (WF w) (WB w))
                             (wte_app WF' WB').

wtews-addr : wte-wsubst J LT ([w] wte_addr LTI) (wte_addr LTI).
wtews-lab  : wte-wsubst J LT ([wt] wte_lab LTI) (wte_lab LTI).

wtews-pair : wte-wsubst J LT ([wt] W1 wt) W1' ->
             wte-wsubst J LT ([wt] W2 wt) W2' ->
             wte-wsubst J LT ([wt] wte_pair (W1 wt) (W2 wt))
                             (wte_pair W1' W2').

wtews-fst : wte-wsubst J LT ([wt] WW wt) WW' ->
            wte-wsubst J LT ([wt] wte_fst (WW wt)) (wte_fst WW').

wtews-snd : wte-wsubst J LT ([wt] WW wt) WW' ->
            wte-wsubst J LT ([wt] wte_snd (WW wt)) (wte_snd WW').

wtews-here : wte-wsubst J LT ([wt] WW wt) WW' ->
             wte-wsubst J LT ([wt] wte_here (WW wt)) (wte_here WW').

wtews-hold : wte-wsubst J LT ([wt] WW wt) WW' ->
             wte-wsubst J LT ([wt] wte_hold (WW wt)) (wte_hold WW').

wtews-held : wte-wsubst J LT ([wt] WW wt) WW' ->
             wte-wsubst J LT ([wt] wte_held (WW wt) VAL) (wte_held WW' VAL).

wtews-box :  ({wo : world}
              {wtw : {X : nat}{S : wslist X}
                      welltyped_world S wo }
```

```
                  {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
                         {EX : extends s s'}
                         weaken-wtw EX (wtw Y s) (wtw Y s')}
               wte-wsubst J LT ([w] W w wo wtw) (W' wo wtw)) ->
            wte-wsubst J LT ([w] wte_box (W w)) (wte_box W').

wtews-lam : ({xx : exp} {ww : {X : nat} {SS : wslist X}
                              welltyped_exp SS xx A' _}
               % substituting into this variable results in the variable
               {thm :
                  {X : nat} {SS : wslist X} {JJ : nat} {LLT : JJ < X}
                  wte-wsubst JJ LLT ([_] ww X SS) (ww X SS)}
               wte-wsubst J LT ([wt] WW xx ww wt) (WW' xx ww)) ->
            wte-wsubst J LT ([wt] wte_lam ([xx][ww] WW xx ww wt))
                 (wte_lam ([xx][ww] WW' xx ww)).

wtews-letdia :
           wte-wsubst J LT ([wt] M wt) M' ->
           % essentially same as lambda case + box case
            ({wo : world}
                  {wtw : {X : nat}{S : wslist X}
                         welltyped_world S wo }
                  {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
                         {EX : extends s s'}
                         weaken-wtw EX (wtw Y s) (wtw Y s')}

               {xx : exp} {wtt : {X : nat} {SS : wslist X}
                                 welltyped_exp SS xx A' wo}
               {thm :
                  {X : nat} {SS : wslist X} {JJ : nat} {LLT : JJ < X}
                  wte-wsubst JJ LLT ([_] wtt X SS) (wtt X SS)}
               wte-wsubst J LT ([wt] N wo wtw xx wtt wt)
                               (N' wo wtw xx wtt)) ->
           wte-wsubst J LT ([wt] wte_letdia (M wt)
                 ([ww : world][wwt][xx : exp][wtt]
                    N ww wwt xx wtt wt))
                 (wte_letdia M' ([ww][wwt][xx][wtt] N' ww wwt xx wtt)).

wtews-leta :
           wte-wsubst J LT ([wt] M wt) M' ->
            ({xx : exp} {wtt : {X : nat} {SS : wslist X}
                       welltyped_exp SS xx A' W'}
               {thm :
                  {X : nat} {SS : wslist X} {JJ : nat} {LLT : JJ < X}
                  wte-wsubst JJ LLT ([_] wtt X SS) (wtt X SS)}
               wte-wsubst J LT ([wt] N xx wtt wt)
                               (N' xx wtt)) ->
           wte-wsubst J LT ([wt] wte_leta (M wt)
                 ([xx : exp][wtt]
                    N xx wtt wt))
                 (wte_leta M' ([xx][wtt] N' xx wtt)).

wtews-letcc : ({uu : cexp} {ww : {X : nat} {SS : wslist X}
                                 welltyped_cexp SS uu A' W'}
                 wte-wsubst J LT ([wt] WW uu ww wt) (WW' uu ww)) ->
              wte-wsubst J LT ([wt] wte_letcc ([uu][ww] WW uu ww wt))
                   (wte_letcc ([uu][ww] WW' uu ww)).

val-csubst : ({u:cexp} value (M u)) -> {C:cexp} value (M C) -> type.
%mode val-csubst +M +C -M'.

- : val-csubst ([u] valbox) _ valbox.
- : val-csubst ([u] (valaddr : value (addr W N))) _ valaddr.
- : val-csubst ([u] vallam) _ vallam.
- : val-csubst ([u] (valpair (VA u) (VB u))) C (valpair VA' VB')
 <- val-csubst VA C VA'
```

```
 <- val-csubst VB C VB'.
- : val-csubst ([u] (valheld (VH u))) C (valheld VH')
 <- val-csubst VH C VH'.

wte-csubst : welltyped_cexp S N A WA ->
             ({u : cexp}{wt : {X : nat} {SS : wslist X} welltyped_cexp SS u A WA}
                welltyped_exp S (M u) B WB) ->
             welltyped_exp S (M N) B WB -> type.
%mode wte-csubst +D1 +D2 -D3.

% subordination problem: val appears to depend on cexp; it can't; use lemma.
wtecs-heldX : val-csubst VAL C VAL' ->
              wte-csubst WTA ([x][w] WW x w) WW' ->
              wte-csubst (WTA : welltyped_cexp _ C _ _) ([x][w] wte_held (WW x w) (VAL x))
                         (wte_held WW' VAL').

% vars are throws
wtecs-throwv : wte-csubst WTA ([x][w] WT x w) WT' ->
               wte-csubst WTA ([x][w] wte_throw (w _ S) (WT x w)) (wte_throw WTA WT').

% (but we might throw to a different var)
wtecs-throw : wte-csubst WTA ([x][w] WT x w) WT' ->
              wte-csubst WTA ([x][w] wte_throw WC (WT x w)) (wte_throw WC WT').
wtecs-abort : wte-csubst WTA WT WT' ->
              wte-csubst WTA ([x][w] wte_abort (WT x w)) (wte_abort WT').
wtecs-get : wte-csubst WTA ([x][w] WT x w) WT' ->
            wte-csubst WTA ([x][w] wte_get WW (WT x w) MOB) (wte_get WW WT' MOB).
wtecs-unbox : wte-csubst WTA ([x][w] WT x w) WT' ->
              wte-csubst WTA ([x][w] wte_unbox (WT x w)) (wte_unbox WT').
wtecs-app : wte-csubst WTA ([x][w] WF x w) WF' ->
            wte-csubst WTA ([x][w] WB x w) WB' ->
            wte-csubst WTA ([u : cexp]
                            [w : {X : nat} {SS : wslist X}
                                 welltyped_cexp SS u A WA]
                           wte_app (WF u w) (WB u w))
                  (wte_app WF' WB').

wtecs-addr : wte-csubst WTA ([x][w] wte_addr LTI) (wte_addr LTI).
wtecs-lab : wte-csubst WTA ([x][w] wte_lab LTI) (wte_lab LTI).
wtecs-pair : wte-csubst WTA ([x][w] W1 x w) W1' ->
             wte-csubst WTA ([x][w] W2 x w) W2' ->
             wte-csubst WTA ([x][w] wte_pair (W1 x w) (W2 x w))
                            (wte_pair W1' W2').
wtecs-fst : wte-csubst WTA ([x][w] WW x w) WW' ->
            wte-csubst WTA ([x][w] wte_fst (WW x w)) (wte_fst WW').
wtecs-snd : wte-csubst WTA ([x][w] WW x w) WW' ->
            wte-csubst WTA ([x][w] wte_snd (WW x w)) (wte_snd WW').
wtecs-here : wte-csubst WTA ([x][w] WW x w) WW' ->
             wte-csubst WTA ([x][w] wte_here (WW x w)) (wte_here WW').
wtecs-hold : wte-csubst WTA ([x][w] WW x w) WW' ->
             wte-csubst WTA ([x][w] wte_hold (WW x w)) (wte_hold WW').

wtecs-box :  ({wo : world}
                {wtw : {X : nat}{S : wslist X}
                         welltyped_world S wo }
               wte-csubst WTA ([x][w] W x w wo wtw) (W' wo wtw)) ->
             wte-csubst WTA ([x][w] wte_box (W x w)) (wte_box W').

wtecs-lam : ({xx : exp} {ww : {X : nat} {SS : wslist X}
                              welltyped_exp SS xx A' W'}
               % substituting into this variable results in the variable
               {thm : {X : nat} {SS : wslist X}
                      {B : typ} {WO : world} {U : cexp}
                      {WTN : welltyped_cexp SS U B WO}
                      wte-csubst WTN ([_][_] ww X SS) (ww X SS)}
               wte-csubst WTA ([u][w] WW xx ww u w) (WW' xx ww)) ->
```

```
            wte-csubst WTA ([u][w] wte_lam ([xx][ww] WW xx ww u w))
                 (wte_lam ([xx][ww] WW' xx ww)).

wtecs-letdia :
           wte-csubst WTA ([x][wt] M x wt) M' ->
           % essentially same as lambda case + box case
            ({ww : world}
               {wwt : {X : nat} {SS : wslist X}
                       welltyped_world SS ww}
               {xx : exp} {wtt : {X : nat} {SS : wslist X}
                                 welltyped_exp SS xx A' ww}
               {thm : {X : nat} {SS : wslist X}
                      {B : typ} {WO : world} {U : cexp}
                      {WTN : welltyped_cexp SS U B WO}
                      wte-csubst WTN ([_][_] wtt X SS) (wtt X SS)}
               wte-csubst WTA ([u][w] N ww wwt xx wtt u w)
                              (N' ww wwt xx wtt)) ->
           wte-csubst WTA ([u][wt] wte_letdia (M u wt)
                 ([ww : world][wwt][xx : exp][wtt]
                    N ww wwt xx wtt u wt))
                 (wte_letdia M' ([ww][wwt][xx][wtt] N' ww wwt xx wtt)).

wtecs-leta : wte-csubst WTA ([x][wt] M x wt) M' ->
              ({xx : exp} {wtt : {X : nat} {SS : wslist X}
                                 welltyped_exp SS xx A' W'}
               {thm : {X : nat} {SS : wslist X}
                      {B : typ} {WO : world} {U : cexp}
                      {WTN : welltyped_cexp SS U B WO}
                      wte-csubst WTN ([_][_] wtt X SS) (wtt X SS)}
               wte-csubst WTA ([u][w] N xx wtt u w)
                              (N' xx wtt)) ->
             wte-csubst WTA ([u][wt] wte_leta (M u wt)
                   ([xx : exp][wtt] N xx wtt u wt))
                   (wte_leta M' ([xx][wtt] N' xx wtt)).

wtecs-letcc : ({uu : cexp} {ww : {X : nat} {SS : wslist X}
                                 welltyped_cexp SS uu A' W'}
                % no need for theorem here.
                wte-csubst WTA ([u][w] WW uu ww u w) (WW' uu ww)) ->
              wte-csubst WTA ([u][w] wte_letcc ([uu][ww] WW uu ww u w))
                   (wte_letcc ([uu][ww] WW' uu ww)).

wtecs-closed : wte-csubst WTA ([x][w] M) M.

val-subst : ({x:exp} value (M x)) -> {N:exp} value (M N) -> type.
%mode val-subst +M +C -M'.

- : val-subst ([x] valbox) _ valbox.
- : val-subst ([x] (valaddr : value (addr W N))) _ valaddr.
- : val-subst ([x] vallam) _ vallam.
- : val-subst ([x] (valpair (VA x) (VB x))) N (valpair VA' VB')
 <- val-subst VA N VA'
 <- val-subst VB N VB'.
- : val-subst ([x] (valheld (VH x))) N (valheld VH')
 <- val-subst VH N VH'.


% substitution lemma allows us to substitute out "magic" hypotheses.
wte-subst : welltyped_exp S N A WA ->
            ({e : exp}{w : {X : nat} {SS : wslist X} welltyped_exp SS e A WA}
               welltyped_exp S (M e) B WB) ->
            welltyped_exp S (M N) B WB -> type.
%mode wte-subst +D1 +D2 -D3.

wtes-closed : wte-subst WTA ([x][w] M) M.
wtes-var : wte-subst (WTA : welltyped_exp S N A WA) ([x][w] w _ S) WTA.
```

```
wtes-heldX : val-subst VAL C VAL' ->
             wte-subst WTA ([x][w] WW x w) WW' ->
             wte-subst (WTA : welltyped_exp _ C _ _)
                       ([x][w] wte_held (WW x w) (VAL x)) (wte_held WW' VAL').
wtes-throw : wte-subst WTA ([x][w] WT x w) WT' ->
             wte-subst WTA ([x][w] wte_throw WC (WT x w)) (wte_throw WC WT').
wtes-abort : wte-subst WTA WT WT' ->
             wte-subst WTA ([x][w] wte_abort (WT x w)) (wte_abort WT').
wtes-get : wte-subst WTA ([x][w] WT x w) WT' ->
           wte-subst WTA ([x][w] wte_get WW (WT x w) MOB) (wte_get WW WT' MOB).
wtes-unbox : wte-subst WTA ([x][w] WT x w) WT' ->
             wte-subst WTA ([x][w] wte_unbox (WT x w)) (wte_unbox WT').
wtes-app :  wte-subst WTA ([x][w] WF x w) WF' ->
            wte-subst WTA ([x][w] WB x w) WB' ->
            wte-subst WTA ([x : exp]
                           [w : {X : nat} {SS : wslist X}
                                welltyped_exp SS x A WA]
                           wte_app (WF x w) (WB x w))
                  (wte_app WF' WB').

wtes-addr : wte-subst WTA ([x][w] wte_addr LTI) (wte_addr LTI).
wtes-lab : wte-subst WTA ([x][w] wte_lab LTI) (wte_lab LTI).
wtes-pair : wte-subst WTA ([x][w] W1 x w) W1' ->
            wte-subst WTA ([x][w] W2 x w) W2' ->
            wte-subst WTA ([x][w] wte_pair (W1 x w) (W2 x w))
                      (wte_pair W1' W2').
wtes-fst :  wte-subst WTA ([x][w] WW x w) WW' ->
            wte-subst WTA ([x][w] wte_fst (WW x w)) (wte_fst WW').
wtes-snd :  wte-subst WTA ([x][w] WW x w) WW' ->
            wte-subst WTA ([x][w] wte_snd (WW x w)) (wte_snd WW').
wtes-here : wte-subst WTA ([x][w] WW x w) WW' ->
            wte-subst WTA ([x][w] wte_here (WW x w)) (wte_here WW').
wtes-hold : wte-subst WTA ([x][w] WW x w) WW' ->
            wte-subst WTA ([x][w] wte_hold (WW x w)) (wte_hold WW').
wtes-held : wte-subst WTA ([x][w] WW x w) WW' ->
            wte-subst WTA ([x][w] wte_held (WW x w) VAL) (wte_held WW' VAL).
wtes-box :  ({wo : world} {wot : {X : nat} {S : wslist X}
                                 welltyped_world S wo}
               wte-subst WTA ([x][w] W x w wo wot) (W' wo wot)) ->
            wte-subst WTA ([x][w] wte_box (W x w)) (wte_box W').

wtes-lam :  ({xx : exp} {ww : {X : nat} {SS : wslist X}
                              welltyped_exp SS xx A' W'}
               % substituting into this variable results in the variable
               {thm : {X : nat} {SS : wslist X}
                      {B : typ} {WO : world} {N : exp}
                      {WTN : welltyped_exp SS N B WO}
                      wte-subst WTN ([_][_] ww X SS) (ww X SS)}
               wte-subst WTA ([x][w] WW xx ww x w) (WW' xx ww)) ->
            wte-subst WTA ([x][w] wte_lam ([xx][ww] WW xx ww x w))
                 (wte_lam ([xx][ww] WW' xx ww)).

wtes-letcc : ({uu : cexp} {ww : {X : nat} {SS : wslist X}
                                welltyped_cexp SS uu A' W'}
                wte-subst WTA ([x][w] WW uu ww x w) (WW' uu ww)) ->
             wte-subst WTA ([x][w] wte_letcc ([uu][ww] WW uu ww x w))
                  (wte_letcc ([uu][ww] WW' uu ww)).

wtes-letdia :
           wte-subst WTA ([x][wt] M x wt) M' ->
           % essentially same as lambda case + box case
            ({ww : world}
               {wwt : {X : nat} {SS : wslist X}
                       welltyped_world SS ww}
               {xx : exp} {wtt : {X : nat} {SS : wslist X}
                                 welltyped_exp SS xx A' ww}
```

```
               {thm : {X : nat} {SS : wslist X}
                      {B : typ} {WO : world} {N : exp}
                      {WTN : welltyped_exp SS N B WO}
                      wte-subst WTN ([_][_] wtt X SS) (wtt X SS)}
               wte-subst WTA ([x][w] N ww wwt xx wtt x w)
                             (N' ww wwt xx wtt)) ->
           wte-subst WTA ([x][wt] wte_letdia (M x wt)
                 ([ww : world][wwt][xx : exp][wtt]
                    N ww wwt xx wtt x wt))
                 (wte_letdia M' ([ww][wwt][xx][wtt] N' ww wwt xx wtt)).

wtes-leta :
           wte-subst WTA ([x][wt] M x wt) M' ->
            ({xx : exp} {wtt : {X : nat} {SS : wslist X}
                               welltyped_exp SS xx A' W'}
               {thm : {X : nat} {SS : wslist X}
                      {B : typ} {WO : world} {N : exp}
                      {WTN : welltyped_exp SS N B WO}
                      wte-subst WTN ([_][_] wtt X SS) (wtt X SS)}
               wte-subst WTA ([x][w] N xx wtt x w)
                             (N' xx wtt)) ->
           wte-subst WTA ([x][wt] wte_leta (M x wt)
                 ([xx : exp][wtt]
                    N xx wtt x wt))
                 (wte_leta M' ([xx][wtt] N' xx wtt)).

%block bsubst :
  some {A' : typ} {W' : world}
  block {xx : exp} {ww : {X : nat} {SS : wslist X} welltyped_exp SS xx A' W'}
        {thm : {X : nat} {SS : wslist X}
               {B : typ} {WO : world} {N : exp}
               {WTN : welltyped_exp SS N B WO}
               wte-subst WTN ([_][_] ww X SS) (ww X SS)}.

%block bsubstu : some {A' : typ} {W' : world}
                 block {uu : cexp} {ww : {X : nat} {SS : wslist X}
                                         welltyped_cexp SS uu A' W'}.
%block bcsubst :
  some {A' : typ} {W' : world}
  block {xx : exp} {ww : {X : nat} {SS : wslist X} welltyped_exp SS xx A' W'}
        {thm : {X : nat} {SS : wslist X}
               {B : typ} {WO : world} {U : cexp}
               {WTN : welltyped_cexp SS U B WO}
               wte-csubst WTN ([_][_] ww X SS) (ww X SS)}.
%block bwsubst' :
  some {A' : typ} {W' : world}
  block {xx : exp} {ww : {X : nat} {SS : wslist X} welltyped_exp SS xx A' W'}
        {thm : {X : nat} {SS : wslist X}
               {JJ : nat} {LLT : JJ < X}
               wte-wsubst JJ LLT ([_] ww X SS) (ww X SS)}.
%block bcsubstu :
  some {A' : typ} {W' : world}
  block {uu : cexp} {ww : {X : nat} {SS : wslist X} welltyped_cexp SS uu A' W'}.
%block bwwt :
  block {w : world} {wtw : {X : nat}{S : wslist X} welltyped_world S w}.
%block bwsubst :
  block {wo : world}
        {wtw : {X : nat}{S : wslist X} welltyped_world S wo }
        {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
           {EX : extends s s'} weaken-wtw EX (wtw Y s) (wtw Y s')}.

% %%%% Big lemmas, mostly about labels.


% need to know that label lookup will succeed.
```

```
label-lemma :
      {W : wdlist N} {S : wslist N} {J : nat} J < N ->
      welltyped_exp S (lab L) A (wconst J) ->
      welltyped_worlds S W ->
      lookupv W J L V VP ->
      welltyped_exp S V A (wconst J) -> type.
%mode label-lemma +W +S +J +LT +WE +WW +L -WV.

ll-findworld : {S  : wslist X} {J  : nat}
               % searching through the following two
               {W'  : wdlist N} {S' : wslist N}
               % for this offset
               {J'  : nat}
               labtype_is S' L A J' ->
               welltyped_worlds' N IDX S S' W' ->
               lookupv W' J' L V VP ->
               % wtw counts up, lookups down. at
               % every iteration we're getting
               % closer to (resp. further from) J
               plus IDX J' J ->
               welltyped_exp S V A (wconst J) -> type.
%mode ll-findworld +S +J +W +S' +J' +LTI +WTW +LV +PLUS -WE.

% as findworld, find a label once we've reached
% the specific world. This is a little simpler

ll-findlab : % storetype -- result must be wellformed in this
             {S : wslist X}
             % the world we're looking in
             {J : nat}
             % trace of subscripting type table and value table, resp
             tt_sub VT OFF A ->
             lookinv VL OFF V VP ->
             % trace of checking the table against its type
             welltyped_datas S VT VL (wconst J) ->
             welltyped_exp S V A (wconst J) -> type.
%mode ll-findlab +S +J +TTS +LIV +WTD -WE.

llfl-search :
     ll-findlab S J TTS LIV WTDS WE ->
     ll-findlab S J (tts_next TTS) (lookinv_next LIV)
         (wtds_one WTDS _) WE.

llfl-found :
     ll-findlab S J tts_found lookinv_found (wtds_one _ WE) WE.

llfw-search :
     ll-findworld S J WDTL STL J' LTIN WTWTL LUVTL PLUS' WE ->
     plus-flip PLUS PLUS' ->
     ll-findworld S J
        (wd// _ _ WDTL) (ws// _ _ STL) (s J')
        (lti_next LTIN)
        (wtws'_one WTWTL _ _)
        (lookupv_next LUVTL) (PLUS : plus IDX (s J') J) WE.

% here we have TTS, which finds that at the offset OFF,
% the value table contains the type A.
% we also have LIV, which finds that at the offset OFF',
% the value table contains the value V.
% the sizes of the two tables are the same, because
% wtws'_one requires them to be.
% OFF and OFF' are actually equal as a result, but LF doesn't
% know this because it doesn't know that minus is determinisic.

llfw-found :
      ll-findworld S IDX=J
```

```
        (wd// _ VL _) (ws// _ VTT _) 0
        (lti_found LTIM TTS)
        (wtws'_one _ _ WTDS)
        (lookupv_found LUVM LIV) PLUS WE
 <- plus-zero PLUS EQP
 <- minus-eq LUVM LTIM EQO
 <- lookinv-resp EQO LIV LIV'
 <- ll-findlab S IDX=J TTS LIV' WTDS WE.

label-lemma-unwrap :
% lt seems unnecessary -- well typedness will ensure
% that the label is in range.
   label-lemma W S J LT
               (wte_lab LTI : welltyped_exp S (lab L) A (wconst J))
               (wtws WTW') LV WE
   <- plus-commutes p0 POO
   <- ll-findworld S J W S J LTI WTW' LV POO WE.

% same, for continuation labels

clabel-lemma :
      {W : wdlist N}
      {S : wslist N}
      {J : nat}
      J < N ->
      clabtype_is S L A J ->
      welltyped_worlds S W ->
      lookupc W J L K ->
      welltyped_cont S K A (wconst J) -> type.
%mode clabel-lemma +W +S +J +LT +WE +WW +L -WC.

cll-findworld : {S  : wslist X} {J  : nat}
                {W'  : wdlist N} {S' : wslist N} {J'  : nat}
                clabtype_is S' L A J' ->
                welltyped_worlds' N IDX S S' W' ->
                lookupc W' J' L K ->
                plus IDX J' J ->
                welltyped_cont S K A (wconst J) -> type.
%mode cll-findworld +S +J +W +S' +J' +CLTI +WTW +LV +PLUS -WK.

cll-findlab : {S : wslist X} {J : nat}
              tt_sub CT OFF A ->
              lookinc CL OFF K ->
              welltyped_conts S CT CL (wconst J) ->
              welltyped_cont S K A (wconst J) -> type.
%mode cll-findlab +S +J +TTS +LIC +WTC -WK.

cllfl-search :
     cll-findlab S J TTS LIC WTCS WC ->
     cll-findlab S J (tts_next TTS) (lookinc_next LIC)
         (wtcs_one WTCS _) WC.

cllfl-found :
     cll-findlab S J tts_found lookinc_found (wtcs_one WTCS WC) WC.

cllfw-search :
     cll-findworld S J WDTL STL J' LTIN WTWTL LUVTL PLUS' WC ->
     plus-flip PLUS PLUS' ->
     cll-findworld S J
        (wd// _ _ WDTL) (ws// _ _ STL) (s J')
        (clti_next LTIN)
        (wtws'_one WTWTL _ _)
        (lookupc_next LUVTL) (PLUS : plus IDX (s J') J) WC.

cllfw-found :
      cll-findworld S IDX=J
```

```
        (wd// CL _ _) (ws// CTT _ _) 0
        (clti_found LTIM TTS)
        (wtws'_one _ WTCS _)
        (lookupc_found LUVM LIV) PLUS WE
 <- plus-zero PLUS EQP
 <- minus-eq LUVM LTIM EQO
 <- lookinc-resp EQO LIV LIV'
 <- cll-findlab S IDX=J TTS LIV' WTCS WE.

clabel-lemma-unwrap :
   clabel-lemma W S J LT (CLTI : clabtype_is S L A J)
               (wtws WTW') LV WC
   <- plus-commutes p0 POO
   <- cll-findworld S J W S J CLTI WTW' LV POO WC.

% need to know that we have a well-typed network after introducing
% a new label.

addlabel-lemma :
     {S : wslist N} {A : typ} {J : nat} J < N ->
     welltyped_exp S V A (wconst J) ->
     welltyped_worlds S W ->
     add_value W J V VP   W' L ->
     % out
     {S' : wslist N}
     extends S S' ->
     labtype_is S' L A J ->
     welltyped_worlds S' W' -> type.

%mode addlabel-lemma +S +A +J +LT +WTE +WTWS +AV  -S' -EXT -LTI -WTWS'.

% we need to trace the generation of S' to provide breadcrumbs later
% the length-j prefix of S and S' is equal, and when we get to the jth
% world, we add type A.

news-trace : {J : nat} {A : typ} {S : wslist N} {S' : wslist N} type.

news-trace-zero :
     news-trace 0 A (ws// CT BT S) (ws// CT (tt// A BT) S).

news-trace-succ :
     news-trace J A S S' ->
     news-trace (s J) A (ws// CT BT S) (ws// CT BT S').

all-newstore :
     % looking through this storetype tail for this offset
     {SL : wslist M} {JL : nat} JL < M ->
     % and will add this type
     {A : typ} add_value WL JL V VP WL' L ->
     welltyped_worlds' M IDX S SL WL ->
     % outputs
     {SL' : wslist M}
     extends SL SL' ->
     labtype_is SL' L A JL ->
     news-trace JL A SL SL' -> type.
%mode all-newstore +SL +JL +LT +A +AV +WT' -SL' -EXT -LTI -NT.

allns-miss :
     all-newstore TTL J LT A AV WT'  TTL' E LTI NT ->
     all-newstore (ws// CT BT TTL) (s J) (lts LT) A
                  (addv_next AV) (wtws'_one WT' _ _)

                  (ws// CT BT TTL')
                  (ex-one E extendst-same extendst-same) (lti_next LTI)
                  (news-trace-succ NT).
```

```
allns-hit :
     minus-self SIZE-1 M0 ->
     ex-id TTL E ->
     all-newstore (ws// CT BT TTL) 0 lt0 A (addv_found (BD : vlist SIZE-1))
                   % this just gets us |BT| = |BD|
                   (wtws'_one _ _ _)
                   (ws// CT (tt// A BT) TTL)
                   (ex-one E extendst-same extendst-cons)
                   (lti_found M0 tts_found) news-trace-zero.

%worlds (bm) (all-newstore _ _ _ _ _ _ _ _ _ _).
%total JL (all-newstore SL JL LT A AV WT'  SL' EXT LTI NT).

% after extending a world with a label, the worlds are still well-typed.
all-still-wtws :
     {S   : wslist N} {S'  : wslist N} {SL  : wslist M}
     {SL' : wslist M} {WL  : wdlist M} {WL' : wdlist M} {JL  : nat}
     welltyped_exp S' V A (wconst J) ->
     welltyped_worlds' M IDX S SL WL ->
     add_value WL JL V VP WL' L ->
     extends S S' ->
     % must know that we haven't changed S except where we did addvalue.
     news-trace JL A SL SL' ->
     plus JL IDX J ->
     extends SL SL' ->
     welltyped_worlds' M IDX S' SL' WL' -> type.
%mode all-still-wtws +S +S' +SL +SL' +WL +WL' +JL +WTE +WT +AV +E +NT +P +EE -WT'.

asw-hit :
    weaken-wtws EXTS WTWS WTWS' ->
    weaken-wtcs EXTS WTCS WTCS' ->
    weaken-wtds EXTS WTDS WTDS' ->
    wte-resp _ _ EQ' WTE WTE' ->
    eq-refl EQ EQ' ->
    plus-zero-eq PLUS EQ ->
    all-still-wtws S S' % nb need to ensure that SL = SL'
                   (ws// CT BT SL) (ws// CT (tt// A BT) SL)
                   (wd// CD BD WL) (wd// CD (v// V VP BD) WL) 0
                   WTE (wtws'_one WTWS WTCS WTDS)
                   (addv_found BD) EXTS news-trace-zero PLUS
                   (ex-one E extendst-same extendst-cons)
                   (wtws'_one WTWS' WTCS'
                    (wtds_one WTDS' WTE')).

asw-skip :
     weaken-wtds EXTS WTD WTD' ->
     weaken-wtcs EXTS WTC WTC' ->
     all-still-wtws S S' SL SL' WL WL' JL WTE WTWS' AV EXTS SF PLUS' E WTW2 ->
     plus-commutes PLUSR' PLUS' ->
     plus-flip PLUSR PLUSR' ->
     plus-commutes PLUS PLUSR ->
     all-still-wtws S S' (ws// CT BT SL) (ws// CT BT SL')
                         (wd// CD BD WL) (wd// CD BD WL') (s JL)
                         WTE (wtws'_one WTWS' WTC WTD)
                         (addv_next AV)
                         EXTS
                         (news-trace-succ SF) PLUS
                         (ex-one E _ _) (wtws'_one WTW2 WTC' WTD').


% strategy for addlabel-lemma: first, recurse to create S' and EXT and LTI.
% these are the *inputs* to the code that gets WTWS.
all-unwrap :
     all-still-wtws S S' S S' W W' J WTE' WTWS' AV EXT NT p0 EXT WTWS2 ->
     weaken-wte EXT WTE WTE' ->
     all-newstore S J LT A AV WTWS'   S' EXT LTI NT ->
```

```
     addlabel-lemma S A J LT WTE (wtws WTWS') AV S' EXT LTI (wtws WTWS2).


% this is almost exactly the same, except we modify the
% continuation table instead of the value table.
addcont-lemma :
    {S : wslist N} {A : typ} {J : nat} J < N ->
    welltyped_cont S K A (wconst J) ->
    welltyped_worlds S W ->
    add_cont W J K W' L ->
    {S' : wslist N} extends S S' -> clabtype_is S' L A J ->
    welltyped_worlds S' W' -> type.
%mode addcont-lemma +S +A +J +LT +WC +WTWS +AC -S' -EX -CLTI -W'.

cnews-trace : {J : nat} {A : typ} {S : wslist N} {S' : wslist N} type.

cnews-trace-zero : cnews-trace 0 A (ws// CT BT S) (ws// (tt// A CT) BT S).

cnews-trace-succ : cnews-trace (s J) A (ws// CT BT S) (ws// CT BT S')
                <- cnews-trace J A S S'.

acl-newstore :
     {SL : wslist M} {JL : nat} JL < M -> {A : typ}
     add_cont WL JL K WL' L -> welltyped_worlds' M IDX S SL WL ->
     {SL' : wslist M} extends SL SL' -> clabtype_is SL' L A JL ->
     cnews-trace JL A SL SL' -> type.
%mode acl-newstore +SL +JL +LT +A +AV +WT' -SL' -EXT -LTI -NT.

aclns-miss :
     acl-newstore TTL J LT A AC WT'  TTL' E LTI NT ->
     acl-newstore (ws// CT BT TTL) (s J) (lts LT) A
                  (addc_next AC) (wtws'_one WT' _ _)
                  (ws// CT BT TTL')
                  (ex-one E extendst-same extendst-same) (clti_next LTI)
                  (cnews-trace-succ NT).

aclns-hit :
     minus-self SIZE-1 M0 ->
     ex-id TTL E ->
     acl-newstore (ws// CT BT TTL) 0 lt0 A (addc_found (CD : clist SIZE-1))
                   (wtws'_one _ _ _)
                   (ws// (tt// A CT) BT TTL)
                   (ex-one E extendst-cons extendst-same)
                   (clti_found M0 tts_found) cnews-trace-zero.

%worlds (bm) (acl-newstore _ _ _ _ _ _ _ _ _ _).
%total JL (acl-newstore SL JL LT A AV WT'  SL' EXT LTI NT).

% after extending a world with a label, the worlds are still well-typed.
acl-still-wtws :
     {S   : wslist N} {S'  : wslist N} {SL  : wslist M}
     {SL' : wslist M} {WL  : wdlist M} {WL' : wdlist M} {JL  : nat}
     welltyped_cont S' K A (wconst J) ->
     welltyped_worlds' M IDX S SL WL ->
     add_cont WL JL K WL' L ->
     extends S S' ->
     % must know that we haven't changed S except where we did addvalue.
     cnews-trace JL A SL SL' ->
     plus JL IDX J ->
     extends SL SL' ->
     welltyped_worlds' M IDX S' SL' WL' -> type.
%mode acl-still-wtws +S +S' +SL +SL' +WL +WL' +JL +WTE +WT +AV +E +NT +P +EE -WT'.


acsw-hit :
    weaken-wtws EXTS WTWS WTWS' ->
```

```
    weaken-wtcs EXTS WTCS WTCS' ->
    weaken-wtds EXTS WTDS WTDS' ->
    wtc-resp _ _ EQ' WTC WTC' ->
    eq-refl EQ EQ' ->
    plus-zero-eq PLUS EQ ->
    acl-still-wtws S S'
                  % nb need to ensure that SL = SL'
                  (ws// CT BT SL) (ws// (tt// A CT) BT SL)
                  (wd// CD BD WL) (wd// (c// K CD) BD WL) 0
                  WTC (wtws'_one WTWS WTCS WTDS)
                  (addc_found CD) EXTS cnews-trace-zero PLUS
                  (ex-one E extendst-cons extendst-same)
                  (wtws'_one WTWS' (wtcs_one WTCS' WTC') WTDS').

acsw-skip :
     weaken-wtds EXTS WTDS WTDS' ->
     weaken-wtcs EXTS WTCS WTCS' ->
     acl-still-wtws S S' SL SL' WL WL' JL WTC WTWS' AV EXTS SF PLUS' E WTW2 ->
     plus-commutes PLUSR' PLUS' ->
     plus-flip PLUSR PLUSR' ->
     plus-commutes PLUS PLUSR ->
     acl-still-wtws S S' (ws// CT BT SL) (ws// CT BT SL')
                         (wd// CD BD WL) (wd// CD BD WL') (s JL)
                         WTC (wtws'_one WTWS' WTCS WTDS)
                         (addc_next AV)
                         EXTS
                         (cnews-trace-succ SF) PLUS
                         (ex-one E _ _) (wtws'_one WTW2 WTCS' WTDS').


acl-unwrap :
     acl-still-wtws S S' S S' W W' J WTC' WTWS' AC EXT NT p0 EXT WTWS2 ->
     weaken-wtc EXT WTC WTC' ->
     acl-newstore S J LT A AC WTWS'   S' EXT CLTI NT ->
     addcont-lemma S A J LT WTC (wtws WTWS') AC S' EXT CLTI (wtws WTWS2).


% if subscripting into the type table worked, then
% subscripting into the value table will work.
lprog-sub : {VT : typetable (s N3)} {VD : vlist (s N3)}
            tt_sub VT OFF A -> lookinv VD OFF M VAL -> type.
%mode lprog-sub +VT +VD +TTS -LIV.

lprog-sub-next :
     lprog-sub VT' VD' TTS' LIV' ->
     lprog-sub (tt// _ VT') (v// _ _ VD')
           (tts_next TTS') (lookinv_next LIV').

lprog-sub-found : lprog-sub (tt// A _) (v// M VAL _) tts_found lookinv_found.

% similar lemma for progress.
% knowing that the label is well-formed, and that all worlds are well formed,
% we just need to know that we can produce a value
label-progress :
           labtype_is SL L A J ->
           welltyped_worlds' REST IDX S SL WL ->
           lookupv WL J L V VP -> type.
%mode label-progress +LTI +WT -LUV.

label-progress-next :
           label-progress LTI WTWS LUV ->
           label-progress (lti_next LTI) (wtws'_one WTWS _ _)
                          (lookupv_next LUV).

label-progress-found :
           lprog-sub VT VD TTS LIV ->
```

```
           label-progress (lti_found (MIN : minus _ _ OFF) TTS)
                          (wtws'_one _ _
                                (DTS : welltyped_datas S VT VD _))
                          (lookupv_found MIN LIV).

% same for continuation labels
clprog-sub :
     {CT : typetable (s N3)}
     {CD : clist (s N3)}
     tt_sub CT OFF A ->
     lookinc CD OFF K -> type.
%mode clprog-sub +VT +VD +TTS -LIC.

clprog-sub-next :
     clprog-sub CT' CD' TTS' LIC' ->
     clprog-sub (tt// _ CT') (c// _ CD')
           (tts_next TTS') (lookinc_next LIC').

clprog-sub-found : clprog-sub (tt// A _) (c// K _) tts_found lookinc_found.

clabel-progress :
           clabtype_is SL L A J ->
           welltyped_worlds' REST IDX S SL WL ->
           lookupc WL J L K -> type.
%mode clabel-progress +CLTI +WT -LUC.

clabel-progress-next :
           clabel-progress CLTI WTWS LUC ->
           clabel-progress (clti_next CLTI) (wtws'_one WTWS _ _)
                          (lookupc_next LUC).

clabel-progress-found :
           clprog-sub CT CD TTS LIC ->
           clabel-progress (clti_found (MIN : minus _ _ OFF) TTS)
                          (wtws'_one _
                                (CTS : welltyped_conts S CT CD _) _)
                          (lookupc_found MIN LIC).

%worlds () (lprog-sub _ _ _ _) (label-progress _ _ _)
           (clprog-sub _ _ _ _) (clabel-progress _ _ _).
%total TTS (lprog-sub _ _ TTS _).
%total L (label-progress L _ _).
%total TTS (clprog-sub _ _ TTS _).
%total L (clabel-progress L _ _).


% some small lemmas for progress
addvalue-progress :
     {W : wdlist N}
     {J : nat}
     J < N ->
     {V : exp} {VAL : value V}
     add_value W J V VAL W' L -> type.
%mode addvalue-progress +W +J +LT +V +VAL -AV.

avp-found : addvalue-progress (wd// CD BD _) 0 lt0 V VAL (addv_found BD).
avp-next : addvalue-progress (wd// _ _ W') (s J') (lts LT') V VAL (addv_next AV')
        <- addvalue-progress W' J' LT' V VAL AV'.

% ditto, continuations
addcont-progress :
     {W : wdlist N}
     {J : nat}
     J < N ->
     {K : cont}
     add_cont W J K W' L -> type.
```

```
%mode addcont-progress +W +J +LT +K -AV.

acp-found : addcont-progress (wd// CD BD _) 0 lt0 K (addc_found CD).
acp-next : addcont-progress (wd// _ _ W') (s J') (lts LT') K (addc_next AC')
        <- addcont-progress W' J' LT' K AC'.

%worlds () (addvalue-progress _ _ _ _ _ _) (addcont-progress _ _ _ _ _).
%total J (addvalue-progress _ J _ _ _ _).
%total J (addcont-progress _ J _ _ _).

% if we index into a world's continuation table, then it is in bounds.
rcont-inbounds : {S : wslist X} clabtype_is S L A J -> J < X -> type.
%mode rcont-inbounds +S +C -LT.

rci-found : rcont-inbounds _ (clti_found _ _) lt0.
rci-next : rcont-inbounds (ws// _ _ TL) (clti_next CLTI) (lts LT)
        <- rcont-inbounds TL CLTI LT.

% or value table...
rval-inbounds : {S : wslist X} labtype_is S L A J -> J < X -> type.
%mode rval-inbounds +S +C -LT.

rvi-found : rval-inbounds _ (lti_found _ _) lt0.
rvi-next : rval-inbounds (ws// _ _ TL) (lti_next LTI) (lts LT)
        <- rval-inbounds TL LTI LT.

%worlds () (rcont-inbounds _ _ _) (rval-inbounds _ _ _).
%total C (rcont-inbounds _ C _).
%total C (rval-inbounds _ C _).


% there's no canonical form for bottom
% to get Twelf to split here, we need to insert an arbitrary case, e.g. abort

notval-abort : value (abort M) ->
            (({N : nat} {NN : network N} {NN' : network N} NN |-> NN')) -> type.
%mode notval-abort +V -F.

no-cf-bottom :
     {V : exp} value V ->
     welltyped_exp S V _|_ W ->
     ({N : nat} {NN : network N} {NN' : network N} NN |-> NN') -> type.
%mode no-cf-bottom +A +B +C -D.

ncb-abort :
  notval-abort V FALSE ->
  no-cf-bottom (abort _) V (wte_abort _) FALSE.

%worlds () (notval-abort _ _) (no-cf-bottom _ _ _ _).
%total V (notval-abort V _).
%total C (no-cf-bottom A B C D).

%%%%
%%%%   Theorems.
%%%%

%%%%% Preservation

progress : {NN : network N} wellformed S NN -> (NN |-> NN') -> type.
%mode progress +NN +WF -STEP.

% cases on evaluating
proge-fst : progress (net W C K (eval (fst M))) _ step_pfst.
proge-snd : progress (net W C K (eval (snd M))) _ step_psnd.
proge-here : progress (net W C K (eval (here M))) _ step_phere.
proge-hold : progress (net W C K (eval (hold M))) _ step_phold.
```

```
proge-pair1 : progress (net W C K (eval (pair M N))) _ step_ppair.
proge-addr : progress (net W C K (eval (addr _ _))) _ step_taddr.
proge-lam : progress (net _ _ _ (eval (lam B))) _ step_tlam.
proge-box : progress (net _ _ _ (eval (box B))) _ step_tbox.
proge-app : progress (net _ _ _ (eval (app M N))) _ step_papp.
proge-unbox : progress (net _ _ _ (eval (unbox M))) _ step_punbox.
proge-letdia : progress (net _ _ _ (eval (letdia M N))) _ step_pletdia.
proge-leta : progress (net _ _ _ (eval (leta M N))) _ step_pleta.
proge-abort : progress (net W J _ (eval (abort M))) _ step_abort.

proge-lab :
      label-progress LTI WTWS LUV
         -> progress (net _ _ _ (eval (lab _)))
               (wf_net _ _ (wt_eval (wte_lab LTI)) _ (wtws WTWS))
               (step_lab LUV).

proge-held :
      progress (net _ _ _ (eval (held V)))
               (wf_net _ _ (wt_eval (wte_held _ VAL)) _ _)
      (step_theld :
       _ |-> net _ _ _ (ret _ (valheld VAL))).

proge-letcc : addcont-progress W J LT K AC ->
              progress (net W J K (eval (letcc M)))
                       (wf_net LT _ _ _ WTWS)
                       (step_rletcc AC).

proge-get : addcont-progress W J LT K AC ->
            progress (net W J K (eval (get (wconst JNEW) M)))
                     (wf_net LT _ _ _ WTWS)
                     (step_get AC).

% cases on returning

% loop to ourselves
progr-done : progress (net W C finish (ret V _)) WF step_done.

% impossible case
% is there a better way to do this? Unlike the other cases,
% Twelf can't seem to figure out on its own that there is no
% canonical form for bottom

progr-fabort :
     no-cf-bottom M V WT RIDICULOUS ->
     progress (net W J fabort (ret M V))
              (wf_net _ _|_ (wt_ret WT) wtc_fabort _)
              (RIDICULOUS _ _ (net W J fabort (ret M V))).

% nb, canonical forms comes free because Twelf is so smart!
progr-ffst : progress (net _ _ (ffst K) (ret (pair M1 _) (valpair VAL1 _)))
              (wf_net _ _ (wt_ret WTP) (wtc_fst _) _)
              step_rfst.

progr-fsnd : progress (net _ _ (fsnd K) (ret (pair _ M2) (valpair _ VAL2)))
              (wf_net _ _ (wt_ret WTP) (wtc_snd _) _)
              step_rsnd.

progr-fpair1 : progress (net _ _ (fpair1 K M) (ret N NV))
              (wf_net _ _ _ _ _)
              step_fpair.

progr-fpair2 : progress (net _ _ (fpair2 K M MV) (ret N NV))
              (wf_net _ _ _ _ _)
              step_rpair.

progr-fapp1 : progress (net _ _ (fapp1 K M) (ret N NV))
```

```
              (wf_net _ _ _ _ _)
              step_fapp.

progr-fapp2 : progress (net _ _ (fapp2 K (lam _) vallam) (ret _ _))
              (wf_net _ _ _ _ _)
              step_rapp.

progr-fletdia : progress (net _ _ (fletdia K N) (ret (addr W L) valaddr))
              _ step_rletdia.

progr-fleta : progress (net _ _ (fleta K N) (ret (held _) (valheld _)))
              _ step_rleta.

progr-funbox : progress (net _ _ (funbox K) (ret (box M) valbox))
              _ step_runbox.

progr-rhold : progress (net _ _ (fhold K) (ret V VAL))
              _ step_rhold.

progr-rhere : addvalue-progress W J LT V VAL AV ->
      progress (net W J (fhere K) (ret V VAL))
        (wf_net LT _ (wt_ret WTA) _ WTWS)
        (step_rhere AV).

progr-rreturn : clabel-progress CLTI WTWS LUC ->
progress (net W J (freturn JNEW L) (ret M V))
 (wf_net LT A (wt_ret WTM)
     (wtc_freturn CLTI MOB) (wtws WTWS))
 (step_rreturn LUC).

proge-throw :
      clabel-progress CLTI WTWS LUC ->
      progress (net W J _ (eval (throw M (caddr (wconst JNEW) L))))
               (wf_net LT _ (wt_eval (wte_throw (wtce_caddr CLTI) WE))
                  _ (wtws WTWS))
               (step_throw LUC).

%%%%% Preservation

preservation : {NN : network N}
               {S : wslist N}
               wellformed S NN ->
               (NN |-> NN') ->
               {S' : wslist N}
               extends S S' ->
               wellformed S' NN' -> type.
%mode preservation +NN +S +WF +STEP -S' -E -WF'.

% by induction on step relation |->
presv-done : ex-id S E -> preservation _ S WF step_done S E WF.

presv-pfst : ex-id S E ->
    preservation _ S (wf_net LT A (wt_eval (wte_fst WE)) WC WTWS)
                   step_pfst S E
                   (wf_net LT (A & B) (wt_eval WE) (wtc_fst WC) WTWS).

presv-psnd : ex-id S E ->
    preservation _ S (wf_net LT B (wt_eval (wte_snd WE)) WC WTWS)
                   step_psnd S E
                   (wf_net LT (A & B) (wt_eval WE) (wtc_snd WC) WTWS).

presv-rfst : ex-id S E ->
    preservation _ S (wf_net LT (A & _) (wt_ret (wte_pair WEA _)) (wtc_fst WC) WTWS)
                   step_rfst S E
                   (wf_net LT A (wt_ret WEA) WC WTWS).
```

```
presv-rsnd : ex-id S E ->
    preservation _ S (wf_net LT (_ & B) (wt_ret (wte_pair _ WEB)) (wtc_snd WC) WTWS)
                  step_rsnd S E
                  (wf_net LT B (wt_ret WEB) WC WTWS).

presv-phere : ex-id S E ->
    preservation _ S (wf_net LT (? A) (wt_eval (wte_here WH)) WC WTWS)
                  step_phere S E
                  (wf_net LT A (wt_eval WH) (wtc_here WC) WTWS).

presv-phold : ex-id S E ->
    preservation _ S (wf_net LT (A at _) (wt_eval (wte_hold WH)) WC WTWS)
                  step_phold S E
                  (wf_net LT A (wt_eval WH) (wtc_hold WC) WTWS).

presv-ppair : ex-id S E ->
    preservation _ S (wf_net LT (A & B) (wt_eval (wte_pair WA WB)) WC WTWS)
                  step_ppair S E
                  (wf_net LT A (wt_eval WA) (wtc_pair1 WB WC) WTWS).

presv-fpair : ex-id S E ->
    preservation _ S (wf_net LT A (wt_ret WA) (wtc_pair1 WB WC) WTWS)
                  step_fpair S E
                  (wf_net LT B (wt_eval WB) (wtc_pair2 WA WC) WTWS).

presv-fapp : ex-id S E ->
    preservation _ S (wf_net LT _ (wt_ret WF) (wtc_app1 WB WC) WTWS)
                  step_fapp S E
                  (wf_net LT A (wt_eval WB) (wtc_app2 WF WC) WTWS).

presv-rpair : ex-id S E ->
    preservation _ S (wf_net LT B (wt_ret WB) (wtc_pair2 WA WK) WTWS)
                  step_rpair S E
                  (wf_net LT (A & B) (wt_ret (wte_pair WA WB)) WK WTWS).

% this case is a bit more complex because we have our own substitution
% theorem; see wwte-lam
presv-rapp : ex-id S E ->
    wte-subst WA WF WS ->
    preservation _ S (wf_net LT A (wt_ret WA) (wtc_app2 (wte_lam WF) WC) WTWS)
                  step_rapp S E
                  (wf_net LT B (wt_eval WS) WC WTWS).

% same here, but even worse because we have to substitute for the expression
% and the world
presv-rletdia : ex-id S E ->
    wte-subst (wte_lab LTI) WN' WE' ->
     % bm
    ({m:exp} {wt : {X : nat} {SS : wslist X} welltyped_exp SS m A (wconst JOTHER)}
       {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
                {ex : extends s s'}
                weaken-wte ex (wt Y s) (wt Y s')}
      wte-wsubst JOTHER LTOTHER ([wwt] WN (wconst JOTHER) wwt m wt) (WN' m wt)) ->
    rval-inbounds S LTI LTOTHER ->
    preservation (net W J (fletdia K N) (ret (addr (wconst JOTHER) L) valaddr))
                 S (wf_net LT (? A) (wt_ret (wte_addr LTI))
                        (wtc_letdia WC WN) WTWS)
                  step_rletdia S E
                  (wf_net LT C (wt_eval WE') WC WTWS).

presv-rleta : ex-id S E ->
    wte-subst WM WN WS ->
    preservation (net W J (fleta K N) (ret (held V) (valheld VP))) S
                 (wf_net LT (A at _) (wt_ret (wte_held WM VAL)) (wtc_leta WC WN) WTWS)
                 step_rleta S E
                 (wf_net LT C (wt_eval WS) WC WTWS).
```

```
presv-runbox : ex-id S E ->
    wte-wsubst J LT (WB (wconst J)) WB' ->
    preservation _ S (wf_net LT (! A) (wt_ret (wte_box WB))
                         (wtc_unbox WC) WTWS)
                   step_runbox S E
                   (wf_net LT A (wt_eval WB') WC WTWS).

presv-tlam : ex-id S E ->
    preservation _ S (wf_net LT (A => B) (wt_eval WA) WC WTWS)
                   step_tlam S E
                   (wf_net LT (A => B) (wt_ret WA) WC WTWS).

presv-tbox : ex-id S E ->
    preservation _ S (wf_net LT (! A) (wt_eval WA) WC WTWS)
                   step_tbox S E
                   (wf_net LT (! A) (wt_ret WA) WC WTWS).

presv-theld : ex-id S E ->
    preservation _ S (wf_net LT (A at W') (wt_eval WA) WC WTWS)
                   step_theld S E
                   (wf_net LT (A at W') (wt_ret WA) WC WTWS).

presv-taddr : ex-id S E ->
    preservation _ S (wf_net LT (? A) (wt_eval WA) WC WTWS)
                   step_taddr S E
                   (wf_net LT (? A) (wt_ret WA) WC WTWS).

presv-papp : ex-id S E ->
    preservation _ S (wf_net LT B (wt_eval (wte_app WF WB)) WC WTWS)
                   step_papp S E
                   (wf_net LT (A => B) (wt_eval WF) (wtc_app1 WB WC) WTWS).

presv-punbox : ex-id S E ->
    preservation _ S (wf_net LT A (wt_eval (wte_unbox WB)) WC WTWS)
                   step_punbox S E
                   (wf_net LT (! A) (wt_eval WB) (wtc_unbox WC) WTWS).

presv-pletdia : ex-id S E ->
    preservation _ S (wf_net LT C (wt_eval (wte_letdia WM WN)) WC WTWS)
                   step_pletdia S E
                   (wf_net LT (? A) (wt_eval WM) (wtc_letdia WC WN) WTWS).

presv-pleta : ex-id S E ->
    preservation _ S (wf_net LT C (wt_eval (wte_leta WM WN)) WC WTWS)
                   step_pleta S E
                   (wf_net LT (A at _) (wt_eval WM) (wtc_leta WC WN) WTWS).

presv-rhold : ex-id S E ->
    preservation (net _ _ _ (ret _ VAL))
                   S (wf_net LT A (wt_ret WTE) (wtc_hold WC) WTWS)
                   step_rhold S E
                   (wf_net LT (A at _) (wt_ret (wte_held WTE VAL)) WC WTWS).

presv-rhere :
    % insertion preserves and creates all sorts of stuff
    weaken-wtc EXT WC WC' ->
    addlabel-lemma S A _ LT WTE WTWS AV  S' EXT LTI WTWS' ->
    preservation _ S (wf_net LT A (wt_ret WTE) (wtc_here WC) WTWS)
                   (step_rhere AV) S' EXT
                   (wf_net LT (? A) (wt_ret (wte_addr LTI)) WC' WTWS').

presv-lab :
    % lookup preserves type
    label-lemma W S J LT WE WTWS LUV WEL ->
    ex-id S E ->
```

```
    preservation _ S
           (wf_net LT A (wt_eval WE) WTC WTWS)
           (step_lab LUV) S E
           (wf_net LT A (wt_ret WEL) WTC WTWS).

% need in vshift: arrows are never mobile,
% so we can conclude anything.
lam-not-mobile : mobile (A => B) ->
                 welltyped_exp S M C W2 ->
                 type.
%mode +{A:typ} +{B:typ} +{MOB:mobile (A => B)}
      +{X:nat} +{S:wslist X} +{M:exp} +{C:typ} +{W2:world}
      -{WT:welltyped_exp S M C W2}
      lam-not-mobile MOB WT.
% no cases
%worlds (bm) (lam-not-mobile _ _).
%total D (lam-not-mobile D _).

% mobile values can be moved between worlds safely
vshift : mobile A ->
         welltyped_exp S M A W1 ->
         value M ->
         {W2 : world}
         welltyped_exp S M A W2 -> type.
%mode vshift +M +WT +V +W2 -WT'.

- : vshift !mob (wte_box WB) valbox _ (wte_box WB).
- : vshift ?mob (wte_addr WH) valaddr _ (wte_addr WH).
- : vshift (&mob Ma Mb) (wte_pair WA WB) (valpair VA VB) W' (wte_pair WA' WB')
 <- vshift Ma WA VA W' WA'
 <- vshift Mb WB VB W' WB'.
- : vshift atmob (wte_held WH VAL) (valheld _) _ (wte_held WH VAL).
- : vshift LM _ vallam _ FALSE
 <- lam-not-mobile LM FALSE.

presv-rreturnbox :
    vshift MOB WM VALUE _ WM' ->
    clabel-lemma W S J LTNEW CLTI WTWS LUC WTC ->
    rcont-inbounds S CLTI LTNEW ->
    ex-id S E ->
    preservation (net _ _ _ (ret _ VALUE)) S
           (wf_net LT A (wt_ret WM) (wtc_freturn CLTI MOB) WTWS)
           (step_rreturn LUC) S E
           (wf_net LTNEW A (wt_ret WM') WTC WTWS).

% throw is rather like the above

presv-throw :
    clabel-lemma W S JNEW LTNEW CLTI (wtws WTWS) LUC   WTC ->
    rcont-inbounds S CLTI LTNEW ->
    ex-id S E ->
    preservation
           (net W JOLD _ (eval (throw M (caddr (wconst JNEW) L)))) S
           (wf_net LT _ (wt_eval (wte_throw (wtce_caddr CLTI) WE)) _
              (wtws WTWS))
           (step_throw LUC) S E
           (wf_net LTNEW _ (wt_eval WE) WTC (wtws WTWS)).

% addition of continuation label preserves stuff
presv-rletcc :
    wte-csubst (wtce_caddr CLTI) WTBOD' WTS ->
    ({c:cexp}
     {wt : {n:nat} {s1:wslist n} welltyped_cexp s1 c A _}
       {thm : {Y : nat} {s : wslist Y} {s' : wslist Y}
              {EX : extends s s'}
              weaken-wtce EX (wt Y s) (wt Y s')}
```

```
       weaken-wte EXT (WTBODY c wt) (WTBOD' c wt)) ->
    weaken-wtc EXT WC WC' ->
    addcont-lemma S A _ LT WC WTWS AC   S' EXT CLTI WTWS' ->
    preservation _ S (wf_net LT A (wt_eval (wte_letcc WTBODY)) WC WTWS)
                  (step_rletcc AC) S' EXT
                  (wf_net LT A (wt_eval WTS) WC' WTWS').

presv-abort : ex-id S EXT ->
    preservation _ S (wf_net LT C
                          (wt_eval (wte_abort WT)) _ WTWS)
                 step_abort S EXT
                 (wf_net LT _|_ (wt_eval WT)
                    wtc_fabort WTWS).

presv-get :
    weaken-wte EXT WT WT' ->
    addcont-lemma S A _ LT WC WTWS AC   S' EXT CLTI WTWS' ->
    preservation _ S
                 (wf_net LT A
                    (wt_eval (wte_get (wtw_const _ LTNEW) WT MOB)) WC WTWS)
                 (step_get AC) S' EXT
                 (wf_net LTNEW A (wt_eval WT')
                    (wtc_freturn CLTI MOB) WTWS').

%%%%
%%%%   Check lemmas and theorems.
%%%%

%worlds (bm) (wte-resp _ _ _ _ _) (wtc-resp _ _ _ _ _) (lookinv-resp _ _ _)
             (lookinc-resp _ _ _).
%total {} (wte-resp _ _ _ _ _).
%total {} (wtc-resp _ _ _ _ _).
%total {} (lookinv-resp _ _ _).
%total {} (lookinc-resp _ _ _).

%worlds (bx | bww) (weaken-wtw _ _ _).
%total W (weaken-wtw _ W _).

%worlds (bx | bww | bu) (weaken-lti _ _ _ _ _) (weaken-clti _ _ _ _ _).
%total LT (weaken-lti  _ _ _ LT _).
%total LT (weaken-clti _ _ _ LT _).

%worlds (bx | bww | bu | bm) (weaken-wtce _ _ _) (weaken-wte _ _ _).
%total W (weaken-wtce _ W _).
%total W (weaken-wte  _ W _).

%worlds (bx | bww | bm) (weaken-wtc _ _ _).
%total W (weaken-wtc _ W _).

%worlds (bm) (weaken-wtds _ _ _) (weaken-wtcs _ _ _) (weaken-wtws _ _ _)
             (all-still-wtws _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
             (acl-still-wtws _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
             (addlabel-lemma _ _ _ _ _ _ _ _ _ _ _)
             (addcont-lemma _ _ _ _ _ _ _ _ _ _ _).
%total W (weaken-wtds _ W _).
%total W (weaken-wtcs _ W _).
%total W (weaken-wtws _ W _).
%total WT (all-still-wtws S S' SL SL' WL WL' JL WTE WT AV E NT P EE WT').
%total WT (acl-still-wtws S S' SL SL' WL WL' JL WTE WT AV E NT P EE WT').
%total [] (addlabel-lemma  _ _ _ _ _ _ _ _ _ _ _).
%total [] (addcont-lemma _ _ _ _ _ _ _ _ _ _ _).

%worlds (bm | bcsubst | bwsubst' | bwsubst | bcsubstu) (wte-wsubst _ _ _ _).
%total D3 (wte-wsubst D1 D2 D3 D4).

%worlds (bm | bcsubst | bwwt | bcsubstu) (val-csubst _ _ _) (wte-csubst _ _ _).
```

```
%total D1 (val-csubst D1 D2 D3).
%total D2 (wte-csubst D1 D2 D3).

%worlds (bm | bsubst | bwwt | bsubstu) (val-subst _ _ _) (wte-subst _ _ _).
%total D1 (val-subst D1 D2 D3).
%total D2 (wte-subst D1 D2 D3).

%worlds (bm) (ll-findlab _ _ _ _ _ _) (ll-findworld _ _ _ _ _ _ _ _ _ _)
   (label-lemma _ _ _ _ _ _ _ _) (cll-findlab _ _ _ _ _ _)
   (cll-findworld _ _ _ _ _ _ _ _ _ _) (clabel-lemma _ _ _ _ _ _ _ _).

%total [TTS LIV WTD] (ll-findlab S J TTS LIV WTD WE).
%total [W' S' J' LTI WTW LV] (ll-findworld S J W' S' J' LTI WTW LV PLUS WE).
%total [WE WW] (label-lemma W S J LT WE WW L WV).

%total [TTS LIV WTD] (cll-findlab S J TTS LIV WTD WE).
%total [W' S' J' LTI WTW LV] (cll-findworld S J W' S' J' LTI WTW LV PLUS WE).
%total [WE WW] (clabel-lemma W S J LT WE WW L WV).

%worlds (bm) (vshift _ _ _ _ _).
%total D (vshift D _ _ _ _).


% the type safety results

%worlds (bm) (preservation NN S WF STEP S E WF).
%total [STEP] (preservation _ _ _ STEP _ _ _).

%worlds () (progress NN WF STEP).
%total [NN WF] (progress NN WF STEP).
```

A.6 Validity

```
%% Soundness and completeness of VS5

world : type.                          %name world W w.
prop : type.                           %name prop A x.

% Propositions

% nontrivial, nonmobile
=> : prop -> prop -> prop.             %infix right 8 =>.
% so that worlds can appear in props
at : prop -> world -> prop.            %infix none 6 at.
sh : (world -> prop) -> prop.
top : prop.
bot : prop.

% Natural deduction

mobile : prop -> type.                 %name mobile M m.

% proofs, valid hypotheses
@ : prop -> world -> type.             %name @ N.
                                       %infix none 1 @.
~ : (world -> prop) -> type.           %name ~ U.

atmob : mobile (A at W).
shmob : mobile (sh F).
topmob : mobile top.
botmob : mobile bot.

% Structural
```

```
get : mobile A -> A @ W -> A @ W'.
put : mobile A -> A @ W -> (A @ W' -> C @ W'') -> C @ W''.

% Universal reasoning
uvar : ~ Af -> (Af W) @ W.
shI  : ({w:world} (Af w) @ w) -> (sh Af @ W).
shE  : sh Af @ W -> (~ Af -> C @ W) -> C @ W.

% Implication
=>I : (A @ W -> B @ W) -> (A => B @ W).
=>E : (A => B @ W) -> A @ W -> B @ W.

% Hybrid
atI  : A @ W -> A at W @ W.
atE  : A at W' @ W -> (A @ W' -> C @ W) -> C @ W.

% Top, bottom
topI : top @ W.
botE : bot @ W -> C @ W.

% Sequent calculus

vhyp : (world -> prop) -> type.         %name vhyp VH vh.
hyp  : prop -> world -> type.           %name hyp H h.
conc : prop -> world -> type.           %name conc D.

%block lo : block {o:world}.
%block lh : some {A:prop} {W:world} block {h:hyp A W}.
%block lv : some {Af:world -> prop} block {v:vhyp Af}.

init : hyp A W -> conc A W.

copy : (hyp (Af W') W' -> conc C W)
       -> (vhyp Af -> conc C W).

topR : conc top W.

botL : hyp bot W -> conc C W'.

=>R : (hyp A W -> conc B W)
       -> conc (A => B) W.
=>L : conc A W
       -> (hyp B W -> conc C U)
       -> (hyp (A => B) W -> conc C U).

atR : conc A W ->
       conc (A at W) W'.
atL : (hyp A W -> conc C W'') ->
       (hyp (A at W) W' -> conc C W'').

shR : ({o:world} conc (Af o) o) -> conc (sh Af) W.
shL : (vhyp Af -> conc C U) ->
      (hyp (sh Af) W -> conc C U).

% metric for types.
% the key thing is that ([w] Af w)
% must be larger than (Af W) for any W
size : type.

size-top : size.
size-bot : size.
size-=> : size -> size -> size.
% in particular, ignore the world here
size-at : size -> size.
size-sh : size -> size.
```

```
% lsize- : size -> size.

has-size  : prop -> size -> type.

has-=> : has-size (A => B) (size-=> Sa Sb)
      <- has-size A Sa
      <- has-size B Sb.

has-at : has-size (A at _) (size-at S) <- has-size A S.
has-sh : has-size (sh ([w] Af w)) (size-sh S) <- ({w: world} has-size (Af w) S).

has-top : has-size top size-top.
has-bot : has-size bot size-bot.

has-lsize : (world -> prop) -> size -> type.

% sneaky move: at some point later twelf wants
% that (lsize- S) = (size-sh S), so let's just
% make them the same constructor (they have the same type)
has-lam : has-lsize ([w] Af w) (size-sh S) % (lsize- S)
       <- ({w} has-size (Af w) S).

can-has-size : {A:prop} has-size A S -> type.
%mode can-has-size +A -H.

- : can-has-size top has-top.
- : can-has-size bot has-bot.
- : can-has-size (A => B) (has-=> Hb Ha)
 <- can-has-size A Ha
 <- can-has-size B Hb.
- : can-has-size (A at W) (has-at H) <- can-has-size A H.
- : can-has-size (sh Af) (has-sh HL) <- ({w} can-has-size (Af w) (HL w)).

can-has-lsize : {Af:world -> prop} has-lsize Af S -> type.
%mode can-has-lsize +Af -H.

- : can-has-lsize ([w] Af w) (has-lam H)
 <- ({w} can-has-size (Af w) (H w)).

cut : {A:prop}
      {S:size}
      has-size A S ->
      conc A W ->
      (hyp A W -> conc C U) ->
      conc C U ->
      type.
%mode cut +A +S +H +D +E -F.

vcut : {Af:world -> prop}
       {S:size}
       (has-lsize Af S) ->
       ({w:world} conc (Af w) w) ->
       (vhyp Af -> conc C W') ->
       conc C W' ->
       type.
%mode vcut +A +S +H +D +E -F.

% right commuting cuts
vr_init : vcut Af _ _ D ([h] init H) (init H).

vr_copy : vcut Af _ Hs D ([h] copy ([h'] E h h') H) (copy F H)
      <- ({h'} vcut Af _ Hs D ([h] E h h') (F h')).

vr_copy_hit : vcut Af _ (has-lam Hs) D
          ([vh] copy ([h' : hyp (Af W1) W1] E vh h') vh) F'
       <- ({h'} vcut Af _ (has-lam Hs) D ([vh : vhyp Af] E vh h') (F h'))
```

```
       <- cut (Af W1) _ (Hs W1) (D W1) F F'.

vr_top : vcut Af _ Hs D ([h] topR) topR.
vr_bot : vcut _ _ _ _ ([h] botL H) (botL H).

vr_=>R : vcut Af _ Hs D ([h] =>R ([h1] E1 h h1)) (=>R ([h1] F1 h1))
      <- ({h1:hyp C1 U} vcut Af _ Hs D ([h] E1 h h1) (F1 h1)).
vr_=>L : vcut Af _ Hs D ([h] =>L (E1 h) ([h2] E2 h h2) H)
                 (=>L F1 ([h2] F2 h2) H)
      <- vcut Af _ Hs D ([h] E1 h) F1
      <- ({h2:hyp B2 U'} vcut Af _ Hs D ([h] E2 h h2) (F2 h2)).
vr_atR : vcut Af _ Hs D ([h] atR (E1 h)) (atR F1)
      <- vcut Af _ Hs D E1 F1.
vr_atL : vcut Af _ Hs D ([h] atL ([h'] E1 h' h) H) (atL F1 H)
      <- ({h'} vcut Af _ Hs D ([h] E1 h' h) (F1 h')).
vr_shL : vcut Af _ Hs D ([h] shL ([vh] E vh h) H) (shL F H)
      <- ({vh} vcut Af _ Hs D ([h] E vh h) (F vh)).

vr_shR : vcut Af _ Hs D ([h] shR ([o] E h o)) (shR F)
      <- ({o} vcut Af _ Hs D ([h] E h o) (F o)).

% Expansion
exp : mobile A -> hyp A W -> conc A W' -> type.   %name exp AM.
%mode +{A:prop} +{M:mobile A}
       +{W:world} +{W':world}
       +{H:hyp A W} -{C:conc A W'}
       exp M H C.

- : exp atmob H (atL ([ha : hyp A W] atR (init ha)) H).
- : exp shmob H (shL ([vh : vhyp Af] shR ([o] copy ([h] init h) vh)) H).
- : exp topmob _ topR.
- : exp botmob H (botL H).

shift : mobile A -> conc A W -> {W':world} conc A W' -> type.
%mode shift +M +H +W' -C.

- : shift M (atL D H) W' (atL D' H)
 <- ({ha} shift M (D ha) W' (D' ha)).

- : shift atmob (atR D) W' (atR D).
- : shift M (init H) W' D <- exp M H D.

- : shift M (copy ([h : hyp (Cf W2) W2] D h) (VH : vhyp Cf)
              : conc A W) W' (copy D' VH)
 <- ({ha} shift M (D ha) W' (D' ha)).

- : shift M (=>L D1 D2 H) W' (=>L D1 D2' H)
 <- ({hb} shift M (D2 hb) W' (D2' hb)).
- : shift shmob (shR D) W' (shR D).
- : shift M (shL ([vh] D vh) H) W' (shL D' H)
 <- ({vh} shift M (D vh) W' (D' vh)).

- : shift topmob _ _ topR.
- : shift _ (botL H) W' (botL H).

% Initial cuts
ci_l : cut A _ _ (init H) ([h] E h) (E H).
ci_r : cut A _ _ D ([h] init h) D.

% Principal cuts
c_=> : cut (A1 => A2) _ (has-=> Hb Ha) (=>R ([h1] D2 h1))
           ([h] =>L (E1 h) ([h2] E2 h h2) h) F
    <- cut (A1 => A2) _ (has-=> Hb Ha) (=>R ([h1] D2 h1)) ([h] E1 h) E1'
    <- ({h2:hyp A2 W}
        cut (A1 => A2) _ (has-=> Hb Ha)
                         (=>R ([h1] D2 h1))
```

```
                      ([h] E2 h h2) (E2' h2))
    <- cut A1 _ Ha E1' ([h1] D2 h1) F1
    <- cut A2 _ Hb F1 ([h2] E2' h2) F.

c_at : cut (A at W) _ (has-at Hs) (atR D) ([h] atL ([h'] E1 h h') h) F
    <- ({h'} cut (A at W) _ (has-at Hs) (atR D) ([h] E1 h h') (E1' h'))
    <- cut A _ Hs D E1' F.

c_sh : cut (sh Af) _ (has-sh Hs) (shR ([w] D w))
                   ([h : hyp (sh Af) W] shL ([vh:vhyp Af] E h vh) h)
                   F'
    <- ({vh:vhyp Af} cut (sh Af) _ (has-sh Hs) (shR D) ([h] E h vh) (F vh))
    <- vcut Af _ (has-lam Hs) D F F'.

c_bot : cut _ _ _ D ([h] botL h) D.

% Right commuting cuts
cr_init : cut A _ _ D ([h] init H) (init H).
cr_copy : cut A _ Hs D ([h] copy ([h'] E h h') H) (copy F H)
       <- ({h'} cut A _ Hs D ([h] E h h') (F h')).

cr_=>R : cut A _ Hs D ([h] =>R ([h1] E1 h h1)) (=>R ([h1] F1 h1))
      <- ({h1:hyp C1 U} cut A _ Hs D ([h] E1 h h1) (F1 h1)).
cr_=>L : cut A _ Hs D ([h] =>L (E1 h) ([h2] E2 h h2) H)
                  (=>L F1 ([h2] F2 h2) H)
      <- cut A _ Hs D ([h] E1 h) F1
      <- ({h2:hyp B2 U'} cut A _ Hs D ([h] E2 h h2) (F2 h2)).


cr_atR : cut A _ Hs D ([h] atR (E1 h)) (atR F1)
      <- cut A _ Hs D E1 F1.
cr_atL : cut A _ Hs D ([h] atL ([h'] E1 h' h) H) (atL F1 H)
      <- ({h'} cut A _ Hs D ([h] E1 h' h) (F1 h')).
cr_shL : cut A _ Hs D ([h] shL ([vh] E vh h) H) (shL F H)
      <- ({vh} cut A _ Hs D ([h] E vh h) (F vh)).
cr_shR : cut A _ Hs D ([h] shR ([o] E h o)) (shR F)
      <- ({o} cut A _ Hs D ([h] E h o) (F o)).
cr_topR : cut _ _ _ _ ([h] topR) topR.
cr_botL : cut _ _ _ _ ([h] botL H) (botL H).

% Left commuting cuts
cl_copy : cut A _ Hs (copy D H) E (copy F H)
       <- ({h} cut A _ Hs (D h) E (F h)).
cl_=>L : cut A _ Hs (=>L D1 ([h2] D2 h2) H) ([h] E h)
                    (=>L D1 ([h2] F2 h2) H)
      <- ({h2:hyp B2 U'} cut A _ Hs (D2 h2) ([h] E h) (F2 h2)).
cl_atL : cut A _ Hs (atL ([h'] D1 h') H) ([h] E h) (atL ([h'] F1 h') H)
      <- ({h'} cut A _ Hs (D1 h') ([h] E h) (F1 h')).
cl_shL : cut A _ Hs (shL D H) E (shL F H)
      <- ({vh} cut A _ Hs (D vh) E (F vh)).
cl_botL : cut A _ Hs (botL H) _ (botL H).

%worlds (lo) (can-has-size _ _) (can-has-lsize _ _).
%total A (can-has-size A _).
%total A (can-has-lsize A _).

%worlds (lo | lh | lv) (exp _ _ _) (shift _ _ _ _) (vcut _ _ _ _ _ _) (cut _ _ _ _ _ _).
%total M (exp M _ _).
%total D (shift _ D _ _).
%total {(S S') {(D D') (E E')}} (cut A S _ D E _) (vcut Af S' _ D' E' _).

%%% Translation ND to SEQ

ndseq : A @ W -> conc A W -> type.      %name ndseq R r.
%mode ndseq +N -D.
uhyp : ~ Af -> ({w:world} conc (Af w) w) -> type.
```

```
%mode uhyp +U -VH.

% uvars
ns_u : ndseq (uvar U) (C _)
    <- uhyp U C.

% shamrock
ns_shI : ndseq (shI Vf) (shR D)
      <- ({w} ndseq (Vf w) (D w)).

ns_shE : ndseq (shE M ([u : ~ Af] N u)) F
      <- ndseq M D
      <- ({u : ~ Af}{vh : vhyp Af}
            uhyp u ([w] copy ([h] init h) vh) -> ndseq (N u) (E vh))
      <- can-has-size (sh Af) Hs
      <- cut (sh Af) _ Hs D
         ([h : hyp (sh Af) W] shL ([vh : vhyp Af] E vh) h) F.

% Implication
ns_=>I : ndseq (=>I ([u1] N2 u1)) (=>R ([h1] D2 h1))
      <- ({u1:A1 @ W} {h1:hyp A1 W}
            ndseq u1 (init h1) -> ndseq (N2 u1) (D2 h1)).

ns_=>E : ndseq (=>E N2 N1) D
      <- ndseq N2 D'
      <- ndseq N1 D1
      <- can-has-size (A1 => A2) Hs
      <- cut (A1 => A2) _ Hs D' ([h] =>L D1 ([h2] init h2) h) D.

% Hybrid
ns_atI : ndseq (atI N) (atR D)
      <- ndseq N D.
ns_atE : ndseq (atE N1 ([u : A @ W] N2 u)) D
      <- ndseq N1 D1'
      <- ({u}{h} ndseq u (init h) -> ndseq (N2 u) (D2 h))
      <- can-has-size (A at W) Hs
      <- cut (A at W) _ Hs D1' ([h] atL ([h'] D2 h') h) D.

% Top and bottom
ns_top : ndseq (topI) topR.
ns_bot : ndseq (botE M) F
      <- ndseq M D
      <- can-has-size bot Hs
      <- cut bot _ Hs D ([h] botL h) F.

% Structural
ns_get : ndseq (get MOB N : A @ W) (F : conc A W)
      <- ndseq N (D : conc A W')
      <- shift MOB D W F.
ns_put : ndseq (put MOB M ([u:A @ W'] (N u) : C @ W'')) (F : conc C W'')
      <- ndseq M (D : conc A W)
      <- ({u}{h} ndseq u (init h) -> ndseq (N u) (E h))
      <- shift MOB D W' D'
      <- can-has-size A Hs
      <- cut A _ Hs D' E F.

%block luhs : some {A:prop} {W:world}
              block {u:A @ W} {h:hyp A W} {r:ndseq u (init h)}.
%block lvhs : some {Af:world -> prop}
              block {u : ~ Af}{vh : vhyp Af}
                    {r:uhyp u ([w] copy ([h] init h) vh)}.

%worlds (lo | luhs | lvhs) (ndseq _ _) (uhyp _ _).
%total (N U) (ndseq N _) (uhyp U _).

%%% Translation SEQ to ND
```

```
seqnd  : conc A W -> A @ W -> type.
hypnd  : hyp A W -> A @ W -> type.
vhypnd : vhyp Af -> ~ Af -> type.
%mode seqnd +D -N.
%mode hypnd +H -N.
%mode vhypnd +V -U.

% Init
sn_init : seqnd (init H) N
       <- hypnd H N.

% Hybrid
sn_atR : seqnd (atR D : conc (A at W') W)
               (get atmob (atI N) : (A at W') @ W)
      <- seqnd (D : conc A W') (N : A @ W').

sn_atL : seqnd (atL ([h] D h) H) (atE (get atmob N1) ([u] N2 u))
      <- ({h : hyp A W}{u : A @ W}
    hypnd h u -> seqnd (D h) (N2 u))
      <- hypnd H N1.

% Implication
sn_=>R : seqnd (=>R ([h] D h)) (=>I ([u : A @ W] N u))
      <- ({h:hyp A W} {u:A @ W}
            hypnd h u ->
           seqnd (D h) (N u)).

sn_=>L : seqnd (=>L (D1 : conc A W) ([h2 : hyp B W] D2 h2) H)
           (N2 (=>E N M))
      <- seqnd D1 M
      <- ({h2:hyp B W} {u2:B @ W} hypnd h2 u2 ->
            seqnd (D2 h2) (N2 u2))
      <- hypnd H N.

% Shamrock
sn_shR : seqnd (shR D) (shI N)
      <- ({w} seqnd (D w) (N w)).

sn_shL : seqnd (shL ([vh] Dc vh) H) (shE (get shmob M) N)
      <- ({vh}{u} vhypnd vh u ->
            seqnd (Dc vh) (N u))
      <- hypnd H M.

sn_copy : seqnd (copy ([h] D h) VH) (M (uvar U))
       <- ({h}{u} hypnd h u -> seqnd (D h) (M u))
       <- vhypnd VH U.

sn_topR : seqnd topR topI.
sn_botL : seqnd (botL H) (botE (get botmob M))
       <- hypnd H M.

%block lhut : some {A:prop} {W:world}
              block {h:hyp A W} {u:A @ W} {_:hypnd h u}.
%block lhvt : some {Af:world -> prop}
              block {h:vhyp Af} {u: ~ Af} {_:vhypnd h u}.
%block lx : some {A:prop}{W:world} block {_ : A @ W}.
%worlds (lo | lhut | lhvt | lx) (seqnd D N) (hypnd H N') (vhypnd _ _).

%total (D H V) (seqnd D N) (hypnd H N') (vhypnd V _).
```

A.7 Operational semantics and type safety for validity

```
%% Operational semantics and type safety for VS5
%% Thanks: Rob Simmons, Jason Reed, Dan Licata

world : type.                           %name world W w.
prop : type.                            %name prop A x.

% Propositions

=> : prop -> prop -> prop.              %infix right 8 =>.
at : prop -> world -> prop.             %infix none 6 at.
sh : (world -> prop) -> prop.
top : prop.

% Natural deduction

mobile : prop -> type.                  %name mobile M m.

% Proofs, values, valid hypotheses
@ : prop -> world -> type.              %name @ N.
                                        %infix none 1 @.
/ : prop -> world -> type.              %name / V.
                                        %infix none 1 /.
~ : (world -> prop) -> type.            %name ~ U.

atmob : mobile (A at W).
shmob : mobile (sh F).
topmob : mobile top.

% Structural
get : mobile A -> A @ W -> A @ W'.
put : mobile A -> A @ W -> (~ ([w] A) -> C @ W) -> C @ W.
val : A / W -> A @ W.

% Universal reasoning
uvar : ~ Af -> (Af W) / W.
shI  : ({w:world} (Af w) / w) -> (sh Af / W).
shE  : sh Af @ W -> (~ Af -> C @ W) -> C @ W.

% Implication
=>I : (A / W -> B @ W) -> (A => B / W).
=>E : (A => B @ W) -> A @ W -> B @ W.

% Hybrid
atI  : A @ W -> A at W @ W.
atIv : A / W' -> A at W' / W.
atE  : A at W' @ W -> (A / W' -> C @ W) -> C @ W.

% True
topI : top / W.

% Valid substitution into expressions, values
vsubst : ({w} (Af w) / w) -> (~ Af -> C @ W) -> C @ W -> type.
%mode vsubst +H +N -N'.
vsubstv : ({w} (Af w) / w) -> (~ Af -> C / W) -> C / W -> type.
%mode vsubstv +H +N -N'.

% for variables; closed to uvars
vv-closed : vsubstv _ ([_] V) V.
vv-top : vsubstv _ ([_] topI) topI.
vv-uvar-hit  : vsubstv Vf ([x] uvar x) (Vf _).
vv-uvar-miss : vsubstv _ ([x] uvar Y) (uvar Y).
vv-at : vsubstv Vf ([x] atIv (V x)) (atIv V')
     <- vsubstv Vf V V'.
vv-lam : vsubstv Vf ([x] =>I ([y] E x y)) (=>I ([y] E' y))
```

```
      <- ({y} vsubst Vf ([x] E x y) (E' y)).
vv-sh : vsubstv Vf ([x] shI ([w] Vf2 x w)) (shI Vf2')
     <- ({w} vsubstv Vf ([x] Vf2 x w) (Vf2' w)).

vs-closed : vsubst _ ([x] M) M.
vs-app : vsubst Vf ([x] =>E (M x) (N x)) (=>E M' N')
      <- vsubst Vf M M'
      <- vsubst Vf N N'.
vs-leta : vsubst Vf ([x] atE (M x) ([y] N x y)) (atE M' N')
       <- vsubst Vf M M'
       <- ({y} vsubst Vf ([x] N x y) (N' y)).
vs-at : vsubst Vf ([x] atI (M x)) (atI M')
     <- vsubst Vf M M'.
vs-lets : vsubst Vf ([x] shE (M x) ([u] N x u)) (shE M' N')
       <- vsubst Vf M M'
       <- ({u} vsubst Vf ([x] N x u) (N' u)).
vs-val : vsubst Vf ([x] val (V x)) (val V')
      <- vsubstv Vf V V'.
vs-get : vsubst Vf ([x] get MOB (M x)) (get MOB M')
      <- vsubst Vf M M'.
vs-put : vsubst Vf ([x] put MOB (M x) ([y] N x y)) (put MOB M' N')
      <- vsubst Vf M M'
      <- ({y} vsubst Vf ([x] N x y) (N' y)).

vshift : mobile A -> A / W -> {W':world} A / W' -> type.
%mode vshift +M +V +W' -V'.

s-top : vshift topmob topI _ topI.
s-at  : vshift atmob (atIv V) _ (atIv V).
s-sh  : vshift shmob (shI Vf) _ (shI Vf).

% We cannot prove
%  extract-lam' : (A => B) / W -> (A / W -> B @ W) -> type
%  directly. If Twelf splits on the first argument, then
%  it sees that =>I and uvar can apply, but it is unable
%  to express the extra unification constraint that
%     (A => B) = Af W.
%  Since it cannot resolve the constraint, coverage checking
%  fails. This is unfortunate because, in a world (context) where
%  there are no ~ hypotheses, like the one we use, the
%  constraint does not matter---we would be able to eliminate
%  the uvar case if we simply forged ahead ignoring the constraint.
%  The solution is to internalize the constraint using the
%  judgment eq. If we prevent Twelf from splitting on this input,
%  then it can eliminate the uvar case because of the regular world,
%  and then it can eliminate other constructors like shI and atIv
%  by subsequently splitting on the eq input.

% extract-lam' : (A => B) / W -> type.
% %mode extract-lam' +M.
% - : extract-lam' (uvar _).

eq : prop -> prop -> type.
eq/ : eq A A.

extract-lam : AB / W -> eq AB (A => B) -> (A / W -> B @ W) -> type.
%mode extract-lam +M +E -L.

% do NOT match against eq/ here, because if we split, game over
- : extract-lam (=>I F) _ F.

extract-at : AW' / W -> eq AW' (A at W') -> A / W' -> type.
%mode extract-at +M +E -L.
- : extract-at (atIv V) _ V.

extract-sh : SA / W -> eq SA (sh Af) -> ({w:world} (Af w) / w) -> type.
```

```
%mode extract-sh +M +E -L.
- : extract-sh (shI Vf) _ Vf.

eval : A @ W -> A / W -> type.
%mode eval +E -V.

eval-val : eval (val V) V.
eval-get : eval (get MOB M') V
        <- eval M' V'
        <- vshift MOB V' _ V.
eval-app : eval (=>E M N) V'
        <- eval M VF
        <- extract-lam VF eq/ F
        <- eval N V
        <- eval (F V) V'.
eval-hold : eval (atI M) (atIv V)
         <- eval M V.
eval-leta : eval (atE M N) V'
         <- eval M AV
         <- extract-at AV eq/ V
         <- eval (N V) V'.
eval-lets : eval (shE M N) V
         <- eval M VS
         <- extract-sh VS eq/ Vf
         <- vsubst Vf N N'
         <- eval N' V.
eval-put : eval (put MOB M N) V
        <- eval M V1
        <- ({w} vshift MOB V1 w (Vf w))
        <- vsubst Vf N N'
        <- eval N' V.

%block w : block {w:world}.
%block x : some {A:prop} {W:world} block {x:A / W}.
%block u : some {Af:world -> prop} block {u:~ Af}.

%worlds (w) (extract-lam _ _ _) (extract-at _ _ _) (extract-sh _ _ _).
%total D (extract-lam D _ _).
%total D (extract-at D _ _).
%total D (extract-sh D _ _).

%worlds (w | x | u) (vsubst _ _ _) (vsubstv _ _ _).
%worlds (w) (eval _ _) (vshift _ _ _ _).

%total D (vshift D _ _ _).
%total (D E) (vsubstv _ D _) (vsubst _ E _).
%covers eval +M -N.
% experimental feature
%partial D (eval D _).
```

A.8 Definitions for modal typed compilation

Each of the signatures in this section requires all of the signatures that precede it.

A.8.1 Forward declarations

Twelf requires that some forward definitions be made so that theorems can be proved with respect to regular worlds in which they will later be used. These forward declarations appear here.

```
% Forward declarations of type families and blocks.

world : type.                                          %name world W w.

% It also becomes necessary to declare modes for some things, even though
% they are not used as logic programs. World is a zero-place relation!

%mode world.

% internal language
ty  : type.                                          %name ty A a.
exp  : type.
val  : type.
vval : type.
ofv  : val -> ty -> world -> type.
ofvv : vval -> (world -> ty) -> type.

% CPS language
ctyp : type.                                          %name ctyp A a.
cval : type.                                          %name cval V v.
cvval : type.                                         %name cvval VV vv.
ttoct : ty -> ctyp -> type.                           %name ttoct TTOCT ttoct.
ttoctf : (world -> ty) -> (world -> ctyp) -> type.    %name ttoctf TTOCTF ttoctf.
cofv  : cval -> ctyp -> world -> type.                %name cofv WV wv.
cofvv : cvval -> (world -> ctyp) -> type.             %name cofvv WVV wvv.

% CPS conversion
tocpsv-  : {WV  : ofv V A W} {CT  : ttoct A CA} {WCV : cofv CV CA W} type.

tocpsvv- : {WV : ofvv V Af} {CT : ttoctf Af CAf} {WCV : cofvv CV CAf} type.

% almost everything is done with respect to hypothetical worlds
%block blockw : block {w : world}.

% when descending under a binding in the source language,
% we'll assume that the variable can be translated to a cps variable
%block blockcvar :
              some {A : ty} {A' : ctyp} {W : world} {CTA : ttoct A A'}
              block
               {x}{xof : ofv x A W}
               {x'}{x'of : cofv x' A' W}
               % how to convert it
               {thm:tocpsv- xof CTA x'of}.

%block blockcvvar :
              some {Af : world -> ty} {Af' : world -> ctyp} {CTA : ttoctf Af Af'}
              block {x}{xof : ofvv x Af}
                    {x'}{x'of : cofvv x' Af'}
                    {thm:tocpsvv- xof CTA x'of}.

% Since we end up having to make worlds declarations for relations
% like cexp, we need these.
%block blockcv  : block {v : cval}.
%block blockcvv : block {v : cvval}.
%block blockwcv  : some {W : world} {CA : ctyp}
                   block {r}{rof : cofv r CA W}.
%block blockwcvv : some {CAf : world -> ctyp}
                   block {v}{vof : cofvv v CAf}.

% some world declarations have to be done very early.
%worlds (blockw | blockcvar | blockwcv | blockwcvv)  (world).
```

A.8.2 External language

```
% The external language of ML5 is just like the internal language,
% but it has some derived connectives like Box and Dia.

tye : type.

ate  : tye -> world -> tye.             %infix none 2 ate.
=>e  : tye -> tye -> tye.               %infix right 8 =>e.
&e : tye -> tye -> tye.                 %infix none 9 &e.
she : (world -> tye) -> tye.
addre : world -> tye.
!e : tye -> tye.
?e : tye -> tye.

expe : type.
vale : type.
vvale : type.

% this is the only form of a valid value
vve : (world -> vale) -> vvale.

% values are expressions, too.
valuee : vale -> expe.
% valid values are values, too.
valide : vvale -> vale.

lame : (vale -> expe) -> vale.
appe : expe -> expe -> expe.
mkpaire : expe -> expe -> expe.
paire : vale -> vale -> vale.
fste : expe -> expe.
snde : expe -> expe.
conste : world -> vale.

helde : vale -> vale.
holde : expe -> expe.
shame : vvale -> vale.
letse : expe -> (vvale -> expe) -> expe.
letae : expe -> (vale -> expe) -> expe.

theree : vale -> vale -> vale.
heree  : expe -> expe.
boxe   : (world -> expe) -> vale.
unboxe : expe -> expe.
% world, address, value
letde : expe -> (world -> vale -> vale -> expe) -> expe.

% give world, address, and remote expression
gete : world -> expe -> expe -> expe.
pute : expe -> (vvale -> expe) -> expe.
localhoste : expe.

% %%%%%%%%%%%%%%%%%%%%%
%    EL typing rules
% %%%%%%%%%%%%%%%%%%%%%

% |- e : t @ w
ofe : expe -> tye -> world -> type.
ofve : vale -> tye -> world -> type.

% x ~ w.t   as hypothesis
ofvve : vvale -> (world -> tye) -> type.

% Mobility
mobilee : tye -> type.
%mode mobilee *A.
```

```
addrMe : mobilee (addre W).
atMe   : mobilee (A ate W).
&Me    : mobilee A -> mobilee B ->
         mobilee (A &e B).
boxMe  : mobilee (!e A).
diaMe  : mobilee (?e A).
shaMe  : mobilee (she A).

% the only form of valid value
vvIe : ({w} ofve (Vf w) (Af w) w) ->
        ofvve (vve Vf) Af.

ofvvalide : ofve (valide V) (Af W) W <- ofvve V Af.
ofvaluee : ofe (valuee V) A W <- ofve V A W.

% rules for shamrock
shIe : ofve (shame Vf) (she Af) W <- ofvve Vf Af.
shEe : ofe M (she A) W ->
        ({x:vvale} ofvve x A ->
           ofe (N x) C W) ->
        ofe (letse M N) C W.

oflocalhoste : ofe localhoste (addre W) W.
addrIe : ofve (conste W) (addre W) W'.

atIve : ofve (helde V) (A ate W') W
     <- ofve V A W'.
atIe : ofe (holde M) (A ate W) W
    <- ofe M A W.
atEe : ofe M (A ate W') W ->
       ({v:vale} ofve v A W' ->
 ofe (N v) C W) ->
       ofe (letae M N) C W.

=>Ie : ofve (lame [x:vale] M x) (A =>e B) W
    <- ({x:vale} ofve x A W ->
ofe (M x) B W).
=>Ee : ofe M1 (A =>e B) W ->
       ofe M2 A W ->
       ofe (appe M1 M2) B W.

&Ive : ofve V1 A W ->
       ofve V2 B W ->
       ofve (paire V1 V2) (A &e B) W.
&Ie : ofe M1 A W ->
      ofe M2 B W ->
      ofe (mkpaire M1 M2) (A &e B) W.

&E1e : ofe (fste M) A W <- ofe M (A &e B) W.
&E2e : ofe (snde M) B W <- ofe M (A &e B) W.

ofpute : mobilee A ->
 ofe M A W ->
         ({u:vvale} ofvve u ([w] A) ->
           ofe (N u) C W) ->
         ofe (pute M N) C W.

ofgete : mobilee A ->
 ofe W'R (addre W') W ->
 ofe M A W' ->
 ofe (gete W' W'R M) A W.

?Ive : ofve Va (addre W') W ->
       ofve V A W' ->
       ofve (theree Va V) (?e A) W.
?Ie : ofe M A W ->
```

```
      ofe (heree M) (?e A) W.
?Ee : ofe M (?e A) W ->
      ({w:world}
       {a:vale}{wa : ofve a (addre w) W}
       {x:vale}{wx : ofve x A w} ofe (N w a x) C W) ->
      ofe (letde M N) C W.

!Ie : ({w:world} ofe (MF w) A w) ->
      ofve (boxe MF) (!e A) W.
!Ee : ofe M (!e A) W ->
      ofe (unboxe M) A W.
```

A.8.3 Internal language and type safety

```
%% S5hybrid with world passing and validity context
%% (internal language for ML5)
%% Tom Murphy VII
%% 11 Jan 2006, 24 Oct 2007

%%% Natural deduction with hybrids and quantification

at  : ty -> world -> ty.            %infix none 2 at.
=>  : ty -> ty -> ty.               %infix right 8 =>.
all : (world -> ty) -> ty.
exists : (world -> ty) -> ty.
& : ty -> ty -> ty.                 %infix none 9 &.
sh : (world -> ty) -> ty.
addr : world -> ty.
unit : ty.

vv : (world -> val) -> vval.

% values are expressions, too.
value : val -> exp.
valid : vval -> val.

held : val -> val.
lam : (val -> exp) -> val.
app : exp -> exp -> exp.
pair : val -> val -> val.
fst : exp -> exp.
snd : exp -> exp.
wlam : (world -> val) -> val.
wapp : exp -> world -> exp.
const : world -> val.
1 : val.
sham : vval -> val.
pack : world -> val -> val.
unpack : exp -> (world -> val -> exp) -> exp.
% could be a derived form, but easy to translate
let : exp -> (val -> exp) -> exp.
% give world, address, and remote expression
get : world -> exp -> exp -> exp.
put : exp -> (vval -> exp) -> exp.
localhost : exp.
lets : exp -> (vval -> exp) -> exp.
leta : exp -> (val -> exp) -> exp.

%% typing judgments for expressions

mobile : ty -> type.
%mode mobile *A.
```

```
addrM : mobile (addr W).
atM  : mobile (A at W).
&M   :  mobile A -> mobile B ->
        mobile (A & B).
allM : ({w} mobile (A w)) -> mobile (all [w] A w).
existsM : ({w} mobile (A w)) -> mobile (exists [w] A w).
shaM : mobile (sh Af).

% |- e : t @ w
of : exp -> ty -> world -> type.
% (others were forward-declared)

vvI : ({w} ofv (Vf w) (Af w) w) -> ofvv (vv Vf) Af.

% containments
ofvalue : of (value V) A W <- ofv V A W.
ofvvalid : ofv (valid V) (Af W) W <- ofvv V Af.

% rules for shamrock
shI : ofv (sham Vf) (sh Af) W <- ofvv Vf Af.

shE : of M (sh A) W ->
      ({x:vval} ofvv x A ->
         of (N x) C W) ->
      of (lets M N) C W.

oflocalhost : of localhost (addr W) W.
addrI : ofv (const W) (addr W) W'.

existsI : {A:world -> ty} % often need this annotation for Twelf's sake
          ofv V (A W') W ->
          ofv (pack W' V) (exists [w] A w) W.
existsE : of M (exists A) W ->
          ({w:world}{v:val} ofv v (A w) W ->
            of (N w v) C W) ->
          of (unpack M N) C W.

atIv : ofv (held V) (A at W') W <- ofv V A W'.
atE : of M (A at W') W ->
      ({v:val} ofv v A W' ->
        of (N v) C W) ->
      of (leta M N) C W.

unitI : ofv 1 unit W.

=>I : ofv (lam [x:val] M x) (A => B) W
     <- ({x:val} ofv x A W ->
                 of (M x) B W).
=>E : of M1 (A => B) W ->
      of M2 A W ->
      of (app M1 M2) B W.

&Iv : ofv V1 A W ->
      ofv V2 B W ->
      ofv (pair V1 V2) (A & B) W.
&E1 : of (fst M) A W <- of M (A & B) W.
&E2 : of (snd M) B W <- of M (A & B) W.

allI : ofv (wlam [w] M w) (all [w] A w) W
    <- ({w:world} ofv (M w) (A w) W).
allE : of (wapp M W') (B W') W
    <- of M (all [w] B w) W.

oflet : of M A W ->
        ({y:val}{ofy : ofv y A W}
           of (N y) C W) ->
```

```
        of (let M N) C W.

ofget : mobile A ->
        of W'R (addr W') W ->
        of M A W' ->
        of (get W' W'R M) A W.

ofput : mobile A ->
        of M A W ->
        ({u:vval}{ofu:ofvv u ([w] A)}
          of (N u) C W) ->
        of (put M N) C W.

% continuation-based operational semantics

ec : type.    %name ec E.

efinish : ec.
% waiting for world addr
eget : ec -> exp -> ec.
% waiting for val
eput : ec -> (vval -> exp) -> ec.
% evaluating remotely, returning to the world
eget2 : ec -> world -> ec.
eapp1 : ec -> exp -> ec.
eapp2 : ec -> val -> ec.
ewapp : ec -> world -> ec.
efst : ec -> ec.
esnd : ec -> ec.
eleta : ec -> (val -> exp) -> ec.
eunpack : ec -> (world -> val -> exp) -> ec.
% waiting for exp to be evaluated
elet : ec -> (val -> exp) -> ec.

% wait for shamrocked value
elets : ec -> (vval -> exp) -> ec.

% E is a continuation expecting A at W
ofec : ec -> ty -> world -> type.    %name ofec WE.

ofec_efinish : ofec efinish A W.
ofec_elets : ofec E C W ->
             ({v:vval} ofvv v A ->
               of (N v) C W) ->
             ofec (elets E N) (sh A) W.

ofec_eput : ofec E C W ->
            ({v:vval} ofvv v ([w] A) ->
              of (N v) C W) ->
            mobile A ->
            ofec (eput E N) A W.

ofec_elet : ofec E C W ->
            ({v:val} ofv v A W ->
              of (N v) C W) ->
            ofec (elet E N) A W.

ofec_eunpack : ({w:world}{v:val} ofv v (A w) W ->
                of (F w v) C W) ->
               ofec E C W ->
               ofec (eunpack E F) (exists A) W.

ofec_eget : mobile A ->
            ofec E A W ->
            of M A W' ->
            ofec (eget E M) (addr W') W.
```

```
ofec_eget2 : mobile A ->
             ofec E A W ->
             ofec (eget2 E W) A W'.

ofec_eapp1 : ofec (eapp1 E M) (A => B) W
          <- ofec E B W
          <- of M A W.

ofec_eapp2 : ofec (eapp2 E (lam F)) A W
          <- ofec E B W
          <- ofv (lam F) (A => B) W.

ofec_ewapp : ofec (ewapp E W') (all [w] A w) W
          <- ofec E (A W') W.

ofec_efst : ofec (efst E) (A & B) W <- ofec E A W.
ofec_esnd : ofec (esnd E) (A & B) W <- ofec E B W.

ofec_eleta : ofec E C W ->
             ({x}{ofx : ofv x A W'} of (N x) C W) ->
             ofec (eleta E N) (A at W') W.

% a value is 'moval' if it has associated
% with it a vval (so it can be "lifted")

moval : world -> val -> vval -> type.

moat : moval _ (held V) (vv [_] held V).
mopair : moval W V1 V1' -> moval W V2 V2' ->
         moval W (pair V1 V2) (vv [_] pair (valid V1') (valid V2')).
moall : ({w:world} moval W (V w) (V' w)) ->
        moval W (wlam V) (vv [_] wlam [w] (valid (V' w))).
moexists : moval W V V' -> moval W (pack W' V) (vv [_] pack W' (valid V')).
moaddr : moval W (const M) (vv [_] const M).
mosh : moval W (sham VV) (vv [_] sham VV).
movv : moval W (valid (vv VV)) VV' <- moval W (VV W) VV'.

% in empty context
wellformed : ec -> exp -> world -> ty -> type.

wf_mach : ofec E A W -> of M A W -> wellformed E M W A.

% then operational semantics

step : world -> ec -> exp   ->   world -> ec -> exp -> type.   %name step S.

step_localhostr : step W E localhost W E (value (const W)).

step_existsp : step W E (unpack M N)  W (eunpack E N) M.
step_existsr : step W (eunpack E N) (value (pack W' V)) W E (N W' V).

step_fstr : step W (efst E) (value (pair V1 V2))  W E (value V1).
step_sndr : step W (esnd E) (value (pair V1 V2))  W E (value V2).
step_fstp : step W E (fst M)  W (efst E) M.
step_sndp : step W E (snd M)  W (esnd E) M.

step_appp : step W E (app M1 M2)  W (eapp1 E M2) M1.
step_app1 : step W (eapp1 E M) (value (lam F))  W (eapp2 E (lam F)) M.
step_app2 : step W (eapp2 E (lam F)) (value X)  W E (F X).

step_wappp : step W E (wapp M W')  W (ewapp E W') M.
step_wapp  : step W (ewapp E W') (value (wlam F))  W E (value (F W')).

step_putp : step W E (put M N)  W (eput E N) M.
step_putr : moval W V V' ->
```

```
            step W (eput E N) (value V)   W E (N V').

% in get, we switch worlds
step_getp : step W E (get W' W'R M)  W (eget E M) W'R.
step_getf : step W  (eget E M) (value (const W'))
                 W' (eget2 E W) M. % saving the return address

step_getr : moval W' V V' ->
            step W' (eget2 E W) (value V)  W E (value (valid V')).

step_letp : step W E (let M N)  W (elet E N) M.
step_letr : step W (elet E N) (value V)  W E (N V).

step_letsp : step W E (lets M N)  W (elets E N) M.
step_letsr : step W (elets E N) (value (sham VV))  W E (N VV).

step_letap : step W E (leta M N)  W (eleta E N) M.
step_letar : step W (eleta E N) (value (held V))  W E (N V).

% we maintain the invt that when returning a value, it is
% not of the form (valid (vv _))
step_vvinst : step W E (value (valid (vv V)))  W E (value (V W)).

% we use self stepping to avoid the extra work of
% handling the finish state as terminal.
% nb this overlaps the previous
step_self : step W efinish (value V)  W efinish (value V).

% lemmas.

% closed values of mobile type are movals.
mobmov : mobile A -> ofv V A W -> moval W V V' -> type.
%mode mobmov +MOB +WV -MOV.

mobmov_addr : mobmov addrM addrI moaddr.
mobmov_at : mobmov atM (atIv _) moat.
mobmov_sh : mobmov shaM (shI _) mosh.
mobmov_& : mobmov MA WA MOA -> mobmov MB WB MOB ->
           mobmov (&M MA MB) (&Iv WA WB) (mopair MOA MOB).

mobmov_all : ({w:world} mobmov (MF w) (WF w) (MOF w)) ->
             mobmov (allM MF) (allI WF) (moall MOF).

mobmov_exists : (mobmov (MF W') WF MOF) ->
                 mobmov (existsM MF)
                        (existsI _ WF : ofv (pack W' _) _ _) (moexists MOF).

mobmov_vvalid : mobmov (MOB : mobile (Af W))
                       (ofvvalid (vvI (D : {w} ofv (Vf w) (Af w) w)) : ofv _ (Af W) W)
                       (movv MOV : moval W _ _)
 <- mobmov MOB (D W) MOV.

% well-typed mobile values are valid values
momo : moval W V V' ->
       ofv V A W ->
       ofvv V' ([w] A) ->
       type.
%mode momo +MV +OV -OV'.

momovv : momo (movv MOB) (ofvvalid (vvI UV)) WVV
      <- momo MOB (UV W) WVV.

momoaddr : momo moaddr addrI (vvI [_] addrI).
momoat : momo moat (atIv M) (vvI [_] atIv M).
momosh : momo mosh (shI WVV) (vvI [_] shI WVV).
```

```
momopair : momo (mopair MV1 MV2) (&Iv OM1 OM2)
               (vvI [w] &Iv (ofvvalid D1) (ofvvalid D2))
        <- momo MV1 OM1 D1
        <- momo MV2 OM2 D2.

momoall : momo (moall MV) (allI OM) (vvI [w'] allI ([w] ofvvalid (D w)))
       <- ({w:world}
           momo (MV w) (OM w) (D w)).

momoexists :
    momo (moexists MOV) (existsI Aw WF) (vvI [w'] (existsI _ (ofvvalid F)))
 <- momo MOV WF F.

% safety theorem
prog : wellformed E M W A ->
       step W E M  W' E' M' ->
       type.
%mode prog +WF -S.

presv : wellformed E M W A ->
        step W E M  W' E' M' ->
        wellformed E' M'  W' A' ->
        type.
%mode presv +WF +S -WF'.

%%%%%%%%%%%%%%%%%%%%
% cases for progress
%%%%%%%%%%%%%%%%%%%%

prog_finish : prog (wf_mach ofec_efinish OF) step_self.

% simple, pushing
prog_getp : prog (wf_mach WE (ofget MOB WR WA)) step_getp.
prog_allEp : prog (wf_mach WE (allE WA)) step_wappp.
prog_=>Ep : prog (wf_mach WE (=>E WF WA)) step_appp.
prog_&E1p : prog (wf_mach WE (&E1 WA)) step_fstp.
prog_&E2p : prog (wf_mach WE (&E2 WA)) step_sndp.
prog_exEp : prog (wf_mach WE (existsE _ _)) step_existsp.
prog_putp : prog (wf_mach WE (ofput MOB WM WN)) step_putp.

% if rhs is validval, instantiate it
prog_vv : prog (wf_mach WE (ofvalue (ofvvalid OV))) step_vvinst.

% if rhs is value, then...
prog_&E1 : prog (wf_mach (ofec_efst WE) (ofvalue (&Iv WA WB))) step_fstr.
prog_&E2 : prog (wf_mach (ofec_esnd WE) (ofvalue (&Iv WA WB))) step_sndr.
prog_=>Ef : prog (wf_mach (ofec_eapp1 WA WE) (ofvalue _)) step_app1.
prog_=>Er : prog (wf_mach (ofec_eapp2 WF WE) (ofvalue _)) step_app2.
prog_allEr : prog (wf_mach (ofec_ewapp WE) (ofvalue _)) step_wapp.
prog_exEr : prog (wf_mach (ofec_eunpack _ _) (ofvalue _)) step_existsr.

prog_putr : prog (wf_mach (ofec_eput WE WN MOB) (ofvalue WV)) (step_putr MOV)
         <- mobmov MOB WV MOV.

prog_getf : prog (wf_mach (ofec_eget MOB WE WF) (ofvalue _)) step_getf.
prog_getr : mobmov MOB WV MOV ->
            prog (wf_mach (ofec_eget2 (MOB : mobile A)
                           (WE : ofec E A W))
                           (ofvalue (WV : ofv V A W'))) (step_getr MOV).

prog_localhostr : prog (wf_mach WE oflocalhost) step_localhostr.

prog_letp : prog (wf_mach WE (oflet _ _)) step_letp.
prog_leta : prog (wf_mach (ofec_elet _ _) (ofvalue WV)) step_letr.

prog_letsp : prog (wf_mach WE (shE _ _)) step_letsp.
```

```
prog_letsfv : prog (wf_mach (ofec_elets _ _) (ofvalue (shI VALID))) step_letsr.

prog_letar : prog (wf_mach (ofec_eleta WE WN) (ofvalue (atIv WV))) step_letar.
prog_letap : prog (wf_mach WE (atE _ _)) step_letap.

%%%%%%%%%%%%%%%%%%%%%%%%
% cases for preservation
%%%%%%%%%%%%%%%%%%%%%%%%

presv_self : presv (wf_mach ofec_efinish OF) step_self (wf_mach ofec_efinish OF).
presv_fstr : presv (wf_mach (ofec_efst WE) (ofvalue (&Iv WA WB))) step_fstr
                   (wf_mach WE (ofvalue WA)).
presv_sndr : presv (wf_mach (ofec_esnd WE) (ofvalue (&Iv WA WB))) step_sndr
                   (wf_mach WE (ofvalue WB)).
presv_getp : presv (wf_mach WE (ofget MOB WFR WT)) step_getp
                   (wf_mach (ofec_eget MOB WE WT) WFR).
presv_getf : presv (wf_mach (ofec_eget MOB WE WT) (ofvalue addrI)) step_getf
                   (wf_mach (ofec_eget2 MOB WE) WT).

presv_getr : momo MOV WV VALID ->
             presv (wf_mach (ofec_eget2 (MOB : mobile A)
                            (WE : ofec E A W)) (ofvalue (WV : ofv _ A W')))
                   (step_getr (MOV : moval _ _ _))
                   (wf_mach WE (ofvalue (ofvvalid VALID))).

presv_putp : presv (wf_mach (WE : ofec E C W) (ofput MOB (WM : of M A W) WN)) step_putp
                   (wf_mach (ofec_eput WE WN MOB) WM).
presv_putr : momo MOV WV VALID ->
             presv (wf_mach (ofec_eput WE WN MOB) (ofvalue WV)) (step_putr MOV)
                   (wf_mach WE (WN _ VALID)).
presv_fstp : presv (wf_mach WE (&E1 WP)) step_fstp
                   (wf_mach (ofec_efst WE) WP).
presv_sndp : presv (wf_mach WE (&E2 WP)) step_sndp
                   (wf_mach (ofec_esnd WE) WP).
presv_appp : presv (wf_mach WE (=>E WF WA)) step_appp
                   (wf_mach (ofec_eapp1 WA WE) WF).
presv_app1 : presv (wf_mach (ofec_eapp1 WA WE) (ofvalue WF)) step_app1
                   (wf_mach (ofec_eapp2 WF WE) WA).
presv_app2 : presv (wf_mach (ofec_eapp2 (=>I WF) WE) (ofvalue WA)) step_app2
                   (wf_mach WE (WF X WA)).
presv_existsp : presv (wf_mach WE (existsE WM WN)) step_existsp
                      (wf_mach (ofec_eunpack WN WE) WM).

presv_existsr : presv (wf_mach (ofec_eunpack WN WE)
                               (ofvalue (existsI _ WA)))
                      (step_existsr : step _ _ (value (pack W V)) _ _ _)
                      (wf_mach WE (WN W V WA)).

presv_wapp : presv (wf_mach (ofec_ewapp WE) (ofvalue (allI WF)))
                   (step_wapp : step _ (ewapp _ W) _ _ _ _)
                   (wf_mach WE (ofvalue (WF W))).

presv_wappp : presv (wf_mach WE (allE WF)) step_wappp
                    (wf_mach (ofec_ewapp WE) WF).

presv_localhost : presv (wf_mach WE oflocalhost) step_localhostr
                        (wf_mach WE (ofvalue addrI)).
presv_letp : presv (wf_mach WE (oflet WM WN)) step_letp
                   (wf_mach (ofec_elet WE WN) WM).
presv_letr : presv (wf_mach (ofec_elet WE WN) (ofvalue WV))
                   step_letr (wf_mach WE (WN _ WV)).
presv_letsp : presv (wf_mach WE (shE WM WN)) step_letsp
                    (wf_mach (ofec_elets WE WN) WM).
presv_letsr : presv (wf_mach (ofec_elets WE WN) (ofvalue (shI WV)))
                    step_letsr (wf_mach WE (WN _ WV)).
presv_letap : presv (wf_mach WE (atE WM WN))
```

```
                    step_letap (wf_mach (ofec_eleta WE WN) WM).
presv_letar : presv (wf_mach (ofec_eleta WE WN) (ofvalue (atIv WV)))
                    step_letar (wf_mach WE (WN _ WV)).
presv_vvinst : presv (wf_mach WE (ofvalue (ofvvalid (vvI WVV)))) step_vvinst
               (wf_mach WE (ofvalue (WVV _))).

%block blockwewe : some {A : ty}{W : world}{W' : world}
                   block {x:exp}{OX : of x A W}{OX' : of x A W'}.

%worlds (blockw | blockwewe) (momo _ _ _).
%total [A B] (momo A B _).

%worlds (blockw | blockwewe) (presv WF S WF').
%total S (presv WF S WF').

%worlds (blockw) (mobmov _ _ _).
%total [WV MOB] (mobmov MOB WV _).

%worlds (blockw) (prog WF S').
%total WF (prog WF S').
```

A.8.4 CPS language and type safety

```
% The ML5 CPS language, its static and dynamic semantics,
% and its proof of safety.

cexp : type.    %name cexp C c.
% forward-declared:
% cval : type.    %name cval V v.
% ctyp : type.    %name ctyp A a.
% cvval : type.   %name cvval VV vv.

cat  : ctyp -> world -> ctyp.             %infix none 2 cat.
% since this is unary, we'll use iterated tuples and
% unit to implement multi-arg functions.
ccont  : ctyp -> ctyp.                     %postfix 8 ccont.
call : (world -> ctyp) -> ctyp.
cexists : (world -> ctyp) -> ctyp.
c& : ctyp -> ctyp -> ctyp.                 %infix none 9 c&.
% representation for a world
caddr : world -> ctyp.
cunit : ctyp.
% shamrock
csh : (world -> ctyp) -> ctyp.

% continuation expressions
cfst : cval -> (cval -> cexp) -> cexp.
csnd : cval -> (cval -> cexp) -> cexp.
clocalhost : (cval -> cexp) -> cexp.

clets : cval -> (cvval -> cexp) -> cexp.
cput : cval -> (cvval -> cexp) -> cexp.
cleta : cval -> (cval -> cexp) -> cexp.

% let is not necessary because of the natural let-style
% of continuation exps
cunpack : cval -> (world -> cval -> cexp) -> cexp.
cwapp : cval -> world -> (cval -> cexp) -> cexp.

% this is the translation of 'get'
cgo : world -> cval -> cexp -> cexp.
% continuations can end with a call to a function
ccall : cval -> cval -> cexp.
chalt : cexp.
```

```
% continuation values
cpair : cval -> cval -> cval.
cheld : world -> cval -> cval.
clam : (cval -> cexp) -> cval.
cconst : world -> cval.
cwlam : (world -> cval) -> cval.
cpack : world -> cval -> cval.
c1 : cval.
% inclusion of cvvals in cvals
cvalid : cvval -> cval.
% internalization into sh modality
ch : cvval -> cval.
% the only vval
cvv : (world -> cval) -> cvval.

% need to redefine this, too
cmobile : ctyp -> type.
%mode cmobile *A.

cmob_& : cmobile A -> cmobile B -> cmobile (A c& B).
cmob_addr : cmobile (caddr W).
cmob_at : cmobile (A cat W).
cmob_all : ({w} cmobile (A w)) -> cmobile (call A).
cmob_exists : ({w} cmobile (A w)) -> cmobile (cexists A).
cmob_sh : cmobile (csh A).


% #################
% Static Semantics
% #################

% Well-formed continuation expressions
cof : cexp -> world -> type.              %name cof WC wc.
%mode cof *C *W.

% values and valid values
% cofv : cval -> ctyp -> world -> type.     %name cofv WV wv.
%mode cofv *A *B *C.
% cofvv : cvval -> (world -> ctyp) -> type. %name cofvv WVV wvv.
%mode cofvv *A *B.

co_halt : cof chalt W.

co_go : cofv VW (caddr W') W ->
        cof C W' ->
        cof (cgo W' VW C) W.

co_put : cmobile A ->
 cofv V A W ->
 ({v} cofvv v ([w] A) ->
    cof (N v) W) ->
 cof (cput V N) W.

co_lets : cofv V (csh Af) W ->
  ({v} cofvv v Af ->
    cof (N v) W) ->
  cof (clets V N) W.

co_leta : cofv V (A cat W') W ->
          ({v} cofv v A W' ->
            cof (N v) W) ->
          cof (cleta V N) W.

co_wapp : cofv V (call A) W ->
```

```
          ({v} cofv v (A W') W ->
             cof (N v) W) ->
          cof (cwapp V W' N) W.

co_unpack :
          cofv V (cexists A) W ->
          ({w}{v} cofv v (A w) W ->
              cof (N w v) W) ->
          cof (cunpack V N) W.

co_localhost : ({v} cofv v (caddr W) W ->
                    cof (K v) W) ->
                cof (clocalhost K) W.

co_call : cofv F (A ccont) W ->
          cofv V A W ->
          cof (ccall F V) W.

co_fst : cofv V (A c& B) W ->
         ({v}{ov : cofv v A W} cof (C v) W) ->
         cof (cfst V C) W.

co_snd : cofv V (A c& B) W ->
         ({v}{ov : cofv v B W} cof (C v) W) ->
         cof (csnd V C) W.

cov_unit : cofv c1 cunit W.

cov_pair : cofv V1 A W ->
           cofv V2 B W ->
           cofv (cpair V1 V2) (A c& B) W.

cov_held : cofv V A W' ->
           cofv (cheld W' V) (A cat W') W.

cov_lam : ({x} cofv x A W ->
               cof (M x) W) ->
          cofv (clam M) (A ccont) W.

cov_const : cofv (cconst W) (caddr W) W'.

cov_wlam : ({w} cofv (V w) (A w) W) ->
           cofv (cwlam V) (call A) W.

cov_pack : {A : world -> ctyp} % sometimes need this annotation
           cofv V (A W') W ->
           cofv (cpack W' V) (cexists A) W.

cov_valid : cofvv VV Af ->
            cofv (cvalid VV) (Af W) W.

cov_ch : cofvv VV Af ->
         cofv (ch VV) (csh Af) W.

covv : ({w} cofv (VF w) (Af w) w) ->
       cofvv (cvv VF) Af.

% boring equality lemmas.
ceqtyp : ctyp -> ctyp -> type.
ceqtyp_ : ceqtyp A A.

% equality is preserved under constructors
ceqtyp_& : ceqtyp A A' -> ceqtyp B B' -> ceqtyp (A c& B) (A' c& B') -> type.
- : ceqtyp_& ceqtyp_ ceqtyp_ ceqtyp_.
%mode ceqtyp_& +A +B -C.
```

```
ceqtyp_cont : ceqtyp A A' -> ceqtyp (A ccont) (A' ccont) -> type.
- : ceqtyp_cont ceqtyp_ ceqtyp_.
%mode ceqtyp_cont +A -C.

ceqtyp_sh : ({w} ceqtyp (A w) (A' w)) -> ceqtyp (csh A) (csh A') -> type.
- : ceqtyp_sh ([w] ceqtyp_) ceqtyp_.
%mode ceqtyp_sh +A -C.

ceqtyp_all : ({w} ceqtyp (A w) (A' w)) ->
    ceqtyp (call A) (call A') -> type.
- : ceqtyp_all ([w] ceqtyp_) ceqtyp_.
%mode ceqtyp_all +A -B.

ceqtyp_exists : ({w} ceqtyp (A w) (A' w)) ->
    ceqtyp (cexists A) (cexists A') -> type.
- : ceqtyp_exists ([w] ceqtyp_) ceqtyp_.
%mode ceqtyp_exists +A -B.

ceqtyp_at : {W:world} ceqtyp A A' -> ceqtyp (A cat W) (A' cat W) -> type.
- : ceqtyp_at _ ceqtyp_ ceqtyp_.
%mode ceqtyp_at +W +A -B.


cofv_resp : cofv C A W -> ceqtyp A A' -> cofv C A' W -> type.
%mode cofv_resp +COF +EQ -COF'.

cofv_resp_ : cofv_resp D ceqtyp_ D.


% dynamic semantics. Actually much simpler than the
% dynamic semantics for the Internal Language.

% mobilize a value. only some can be mobilized.
% (this is like the 'get' operations below)
cmoval : world -> cval -> cvval -> type.
%mode cmoval *W *A *B.

cmoat : cmoval _ (cheld W' V) (cvv [_] cheld W' V).
cmopair : cmoval W V1 V1' -> cmoval W V2 V2' ->
         cmoval W (cpair V1 V2) (cvv [_] cpair (cvalid V1') (cvalid V2')).
cmoall : ({w:world} cmoval W (V w) (V' w)) ->
         cmoval W (cwlam V) (cvv [_] cwlam [w] (cvalid (V' w))).
cmoexists : cmoval W V V' -> cmoval W (cpack W' V) (cvv [_] cpack W' (cvalid V')).
cmoaddr : cmoval W (cconst M) (cvv [_] cconst M).
cmosh : cmoval W (ch VV) (cvv [_] ch VV).
cmovv : cmoval W (cvalid (cvv VV)) VV'
     <- cmoval W (VV W) VV'.

% we need a bunch of canonical-forms lemmata, because of valid
% values. We have a "get" theorem for each canonical form, known to be
% total when the argument has a certain type

cget_pair  : world -> cval ->  cval -> cval -> type.
%mode cget_pair *W *A *B *C.
cgpair-it : cget_pair _ (cpair V1 V2) V1 V2.
cgpair-vv : cget_pair W (cvalid (cvv VV)) V1 V2
         <- cget_pair W (VV W) V1 V2.

cget_ch    : world -> cval ->  cvval -> type.
%mode cget_ch *W *A *B.
cgch-it : cget_ch _ (ch VV) VV.
cgch-vv : cget_ch W (cvalid (cvv VV)) VV'
       <- cget_ch W (VV W) VV'.

cget_held  : world -> cval ->  cval -> type.
%mode cget_held *W *A *C.
```

```
cgheld-it : cget_held _ (cheld W' V) V.
cgheld-vv : cget_held W (cvalid (cvv VV)) V
         <- cget_held W (VV W) V.

cget_pack  : world -> cval ->  world -> cval -> type.
%mode cget_pack *W *A *B *C.
cgpack-it : cget_pack _ (cpack W V) W V.
cgpack-vv : cget_pack W (cvalid (cvv VV))  W' V'
         <- cget_pack W (VV W) W' V'.

cget_wlam  : world -> cval ->  (world -> cval) -> type.
%mode cget_wlam *W *A *B.
cgwlam-it : cget_wlam _ (cwlam V) V.
cgwlam-vv : cget_wlam W (cvalid (cvv VV)) V
         <- cget_wlam W (VV W) V.

cget_const : world -> cval ->  world -> type.
%mode cget_const *W *A *B.
cgconst-it : cget_const _ (cconst W') W'.
cgconst-vv : cget_const W (cvalid (cvv VV)) V
          <- cget_const W (VV W) V.

cget_lam   : world -> cval ->  (cval -> cexp) -> type.
%mode cget_lam *W *A *B.
cglam-it : cget_lam _ (clam C) C.
cglam-vv : cget_lam W (cvalid (cvv VV)) C
        <- cget_lam W (VV W) C.

% stepping relation; indexed by current world
cstep : world -> cexp ->
        world -> cexp -> type.
%mode cstep *AW *A *BW *B.

cs_fst        : cstep W (cfst VP C)  W (C V) <- cget_pair W VP V _.
cs_snd        : cstep W (csnd VP C)  W (C V) <- cget_pair W VP _ V.
cs_localhost  : cstep W (clocalhost C)  W (C (cconst W)).
cs_lets       : cstep W (clets VH C)  W (C V) <- cget_ch W VH V.
cs_put        : cstep W (cput V C)  W (C VV) <- cmoval W V VV.
cs_leta       : cstep W (cleta VA C)  W (C V) <- cget_held W VA V.
cs_unpack     : cstep W (cunpack VE C)  W (C W' V) <- cget_pack W VE W' V.
cs_wapp       : cstep W (cwapp VL W' C)  W (C (V W')) <- cget_wlam W VL V.
cs_go         : cstep W (cgo W' VA C)  W' C <- cget_const W VA W'.
cs_call       : cstep W (ccall VL V)  W (C V) <- cget_lam W VL C.
% self-transition, as usual
cs_halt       : cstep W chalt  W chalt.

% some canonical forms lemmata
cf_pack : cofv V A W -> ceqtyp A (cexists A') -> cget_pack W V  W' V' -> type.
%mode cf_pack +O +E -W.
- : cf_pack (cov_pack _ _) _ cgpack-it.
- : cf_pack ((cov_valid (covv D)) : cofv _ _ W) EQ (cgpack-vv Z)
 <- cf_pack (D W) EQ Z.

cf_const : cofv V A W -> ceqtyp A (caddr W') -> cget_const W V W' -> type.
%mode cf_const +O +E -W.
- : cf_const cov_const _ cgconst-it.
- : cf_const ((cov_valid (covv D)) : cofv _ _ W) EQ (cgconst-vv W')
 <- cf_const (D W) EQ W'.

cf_wlam : cofv V A W -> ceqtyp A (call A') -> cget_wlam W V V' -> type.
%mode cf_wlam +O +E -L.
- : cf_wlam (cov_wlam _) _ cgwlam-it.
- : cf_wlam ((cov_valid (covv D)) : cofv _ _ W) EQ (cgwlam-vv L)
 <- cf_wlam (D W) EQ L.

cf_lam : cofv V A W -> ceqtyp A (A' ccont) -> cget_lam W V C -> type.
```

```
%mode cf_lam +O +E -L.
- : cf_lam (cov_lam _) _ cglam-it.
- : cf_lam ((cov_valid (covv D)) : cofv _ _ W) EQ (cglam-vv L)
 <- cf_lam (D W) EQ L.

cf_held : cofv V A W -> ceqtyp A (A' cat W') -> cget_held W V  V' -> type.
%mode cf_held +O +E -H.
- : cf_held (cov_held _) _ cgheld-it.
- : cf_held ((cov_valid (covv D)) : cofv _ _ W) EQ (cgheld-vv H)
 <- cf_held (D W) EQ H.

cf_pair : cofv V A W -> ceqtyp A (A1 c& A2) -> cget_pair W V  V1 V2 -> type.
%mode cf_pair +O +E -P.
- : cf_pair (cov_pair _ _) _ cgpair-it.
- : cf_pair ((cov_valid (covv D)) : cofv _ _ W) EQ (cgpair-vv P)
 <- cf_pair (D W) EQ P.

cf_ch : cofv V A W -> ceqtyp A (csh Af) -> cget_ch W V VV -> type.
%mode cf_ch +O +E -P.
- : cf_ch (cov_ch _) _ cgch-it.
- : cf_ch ((cov_valid (covv D)) : cofv _ _ W) EQ (cgch-vv VV)
 <- cf_ch (D W) EQ VV.

% lemma: if a value is of mobile type, then it is a
% mobile value (like the canonical forms lemmas above)

cmobmov : cmobile A -> cofv V A W -> cmoval W V V' -> type.
%mode cmobmov +MOB +WV -MOV.

- : cmobmov cmob_at _ cmoat.
- : cmobmov MA WA MOA ->
    cmobmov MB WB MOB ->
    cmobmov (cmob_& MA MB) (cov_pair WA WB) (cmopair MOA MOB).
- : cmobmov cmob_addr _ cmoaddr.
- : cmobmov (cmob_all MF) (cov_wlam WF) (cmoall MOF)
 <- ({w} cmobmov (MF w) (WF w) (MOF w)).
- : cmobmov (cmob_exists MF) (cov_pack _ WF : cofv (cpack W' _) _ _) (cmoexists MOF)
 <- cmobmov (MF W') WF MOF.
- : cmobmov cmob_sh (cov_ch WVV) cmosh.
- : cmobmov (MOB : cmobile (Af W))
            (cov_valid (covv (D : {w} cofv (Vf w) (Af w) w)) : cofv _ (Af W) W)
            (cmovv MOV : cmoval W _ _)
 <- cmobmov MOB (D W) MOV.

% some "inversion" lemmata for preservation
ci_ch : cofv V A W -> ceqtyp A (csh Af) -> cget_ch W V VV -> cofvv VV Af -> type.
%mode ci_ch +O +E +P -D.
- : ci_ch (cov_ch D) _ cgch-it D.
- : ci_ch ((cov_valid (covv D)) : cofv _ _ W) EQ (cgch-vv VV) DD
 <- ci_ch (D W) EQ VV DD.

ci_pair : cofv V A W -> ceqtyp A (A1 c& A2) -> cget_pair W V  V1 V2 ->
          cofv V1 A1 W -> cofv V2 A2 W -> type.
%mode ci_pair +O +E +P -V1 -V2.
- : ci_pair (cov_pair D1 D2) _ cgpair-it D1 D2.
- : ci_pair ((cov_valid (covv D)) : cofv _ _ W) EQ (cgpair-vv P) D1 D2
 <- ci_pair (D W) EQ P D1 D2.

ci_lam : cofv V A W -> ceqtyp A (A' ccont) -> cget_lam W V C ->
         ({x} cofv x A' W -> cof (C x) W) -> type.
%mode ci_lam +O +E +L -D.
- : ci_lam (cov_lam D) _ cglam-it D.
- : ci_lam ((cov_valid (covv D)) : cofv _ _ W) EQ (cglam-vv L) DD
 <- ci_lam (D W) EQ L DD.

ci_wlam : cofv V A W -> ceqtyp A (call A') -> cget_wlam W V V' ->
```

```
          ({w} cofv (V' w) (A' w) W) -> type.
%mode ci_wlam +O +E +L -D.
- : ci_wlam (cov_wlam D) _ cgwlam-it D.
- : ci_wlam ((cov_valid (covv D)) : cofv _ _ W) EQ (cgwlam-vv L) D'
 <- ci_wlam (D W) EQ L D'.

ci_pack : {A': world -> ctyp} cofv V A W -> ceqtyp A (cexists A') ->
          cget_pack W V  W' V' -> cofv V' (A' W') W -> type.
%mode ci_pack +O +A' +E +P -B.
- : ci_pack A' (cov_pack A' B : cofv (cpack W' V') (cexists A') W) EQ
     (cgpack-it : cget_pack W (cpack W' V')  W' V') B.
- : ci_pack A' ((cov_valid (covv D)) : cofv _ _ W) EQ (cgpack-vv Z) B
 <- ci_pack A' (D W) EQ Z B.

ci_held : cofv V A W -> ceqtyp A (A' cat W') -> cget_held W V  V' ->
  cofv V' A' W' -> type.
%mode ci_held +O +E +H -V.
- : ci_held (cov_held D) _ cgheld-it D.
- : ci_held ((cov_valid (covv D)) : cofv _ _ W) EQ (cgheld-vv H) DD
 <- ci_held (D W) EQ H DD.

% well-typed mobile values are valid values
cmomo : cmoval W V V' -> cofv V A W -> cofvv V' ([w] A) -> type.
%mode cmomo +MV +OV -OV'.

- : cmomo (cmovv MOB) (cov_valid (covv UV)) WVV
 <- cmomo MOB (UV W) WVV.
- : cmomo cmosh (cov_ch WVV) (covv [_] cov_ch WVV).
- : cmomo cmoaddr cov_const (covv [_] cov_const).
- : cmomo cmoat (cov_held M) (covv [_] cov_held M).
- : cmomo (cmopair MV1 MV2) (cov_pair WV1 WV2)
           (covv [w] cov_pair (cov_valid D1) (cov_valid D2))
 <- cmomo MV1 WV1 D1
 <- cmomo MV2 WV2 D2.
- : cmomo (cmoall MV) (cov_wlam WV) (covv [w'] cov_wlam ([w] cov_valid (D w)))
 <- ({w} cmomo (MV w) (WV w) (D w)).
- : cmomo (cmoexists MV) (cov_pack Af WF) (covv [w'] cov_pack Af (cov_valid F))
 <- cmomo MV WF F.

% progress
cprog : cof C W -> cstep W C  W' C' -> type.
%mode cprog +COF -CSTEP.

- : cprog co_halt cs_halt.
- : cprog (co_fst WV _) (cs_fst G) <- cf_pair WV ceqtyp_ G.
- : cprog (co_snd WV _) (cs_snd G) <- cf_pair WV ceqtyp_ G.
- : cprog (co_call WV _) (cs_call G) <- cf_lam WV ceqtyp_ G.
- : cprog (co_localhost _) cs_localhost.
- : cprog (co_unpack WV _) (cs_unpack G) <- cf_pack WV ceqtyp_ G.
- : cprog (co_wapp WV _) (cs_wapp G) <- cf_wlam WV ceqtyp_ G.
- : cprog (co_leta WV _) (cs_leta G) <- cf_held WV ceqtyp_ G.
- : cprog (co_lets WV _) (cs_lets G) <- cf_ch WV ceqtyp_ G.
- : cprog (co_put MOB WV _) (cs_put CM) <- cmobmov MOB WV CM.
- : cprog (co_go WV _) (cs_go G) <- cf_const WV ceqtyp_ G.

% preservation
cpresv : cof C W -> cstep W C  W' C' -> cof C' W' -> type.
%mode cpresv +COF +CSTEP -COF'.

- : cpresv co_halt cs_halt co_halt.
- : cpresv (co_go WV WC) (cs_go G) WC.
- : cpresv (co_fst WV WC) (cs_fst G) (WC _ WV1) <- ci_pair WV ceqtyp_ G WV1 _.
- : cpresv (co_snd WV WC) (cs_snd G) (WC _ WV2) <- ci_pair WV ceqtyp_ G _ WV2.
- : cpresv (co_call WVF WVA) (cs_call G) (WC _ WVA) <- ci_lam WVF ceqtyp_ G WC.
- : cpresv (co_wapp WV WC : cof (cwapp _ W' _) W) (cs_wapp G) (WC _ (WV' W'))
 <- ci_wlam WV ceqtyp_ G WV'.
```

```
- : cpresv (co_unpack WV WC) (cs_unpack (G : cget_pack _ _ W' V)) (WC W' V WV')
 <- ci_pack _ WV ceqtyp_ G WV'.
- : cpresv (co_leta WA WC) (cs_leta G) (WC _ WV) <- ci_held WA ceqtyp_ G WV.
- : cpresv (co_put MOB WV WC) (cs_put CM) (WC _ WVV) <- cmomo CM WV WVV.
- : cpresv (co_lets WV WC) (cs_lets G) (WC _ WVV) <- ci_ch WV ceqtyp_ G WVV.
- : cpresv (co_localhost WC) cs_localhost (WC _ cov_const).

% These are required because of overconservativity of Twelf's world subsumption (I think)
% in TOCPS. These are strange, but don't hurt anything.
% This is our last chance to declare worlds before they are auto-frozen by the totality
% assertions below.
%worlds (blockw | blockcv | blockcvv) (cval) (cexp) (cvval).
%worlds (blockw | blockwcv | blockwcvv) (cofv _ _ _) (cof _ _) (cofvv _ _) (cmobile _).

%worlds (blockw) (cf_pair _ _ _) (cf_held _ _ _) (cf_lam _ _ _) (cf_wlam _ _ _)
 (cf_const _ _ _) (cf_pack _ _ _) (cf_ch _ _ _) (ci_held _ _ _ _) (ci_pair _ _ _ _ _)
 (ci_lam _ _ _ _) (ci_wlam _ _ _ _) (ci_pack _ _ _ _ _) (ci_ch _ _ _ _) (cmobmov _ _ _)
 (cmomo _ _ _) (cprog _ _) (cpresv _ _ _).

%total D (cmobmov _ D _).
%total D (cmomo D _ _).
%total D (ci_ch D _ _ _).
%total D (ci_held D _ _ _).
%total D (ci_pack _ D _ _ _).
%total D (ci_pair D _ _ _ _).
%total D (ci_lam D _ _ _).
%total D (ci_wlam D _ _ _).
%total D (cf_pack D _ _).
%total D (cf_pair D _ _).
%total D (cf_ch D _ _).
%total D (cf_held D _ _).
%total D (cf_lam D _ _).
%total D (cf_wlam D _ _).
%total D (cf_const D _ _).
%total D (cprog D _).
%total D (cpresv _ D _).
```

A.8.5 Elaboration relation and static correctness

```
% Elaboration of external language (el.elf) into
% internal language (il.elf). This translation is
% mostly pointwise, except that it converts the
% box and dia connectives into uses of
% quantification and the 'at' modality.

% relates EL types to IL types
ettoit : tye -> ty -> type.
%mode ettoit +A -A'.
ettoitf : (world -> tye) -> (world -> ty) -> type.
%mode ettoitf +Af -Af'.
ettoitf_ : ({w} ettoit (Af w) (Af' w)) ->
           ettoitf Af Af'.

ettoit_& : ettoit A A' ->
           ettoit B B' ->
           ettoit (A &e B) (A' & B').

ettoit_=> : ettoit A A' ->
            ettoit B B' ->
            ettoit (A =>e B) (A' => B').

ettoit_at : ettoit A A' ->
            ettoit (A ate W) (A' at W).
```

```
ettoit_sh : ({w} ettoit (Af w) (Af' w)) ->
            ettoit (she Af) (sh Af').

ettoit_addr : ettoit (addre W) (addr W).

ettoit_! : ettoit A A' ->
           ettoit (!e A) (all [w] ((unit => A') at w)).

ettoit_? : ettoit A A' ->
           ettoit (?e A) (exists [w:world] ((A' at w) & addr w)).

%worlds (blockw) (ettoit _ _) (ettoitf _ _).
%total A (ettoit A _).
%total A (ettoitf A _).

eqtyp : ty -> ty -> type.
eqtyp_ : eqtyp A A.

% these are compatibility cases...
eqtyp_& : eqtyp A A' -> eqtyp B B' -> eqtyp (A & B) (A' & B') -> type.
%mode eqtyp_& +A +B -C.
- : eqtyp_& eqtyp_ eqtyp_ eqtyp_.

eqtyp_=> : eqtyp A A' -> eqtyp B B' -> eqtyp (A => B) (A' => B') -> type.
%mode eqtyp_=> +A +B -C.
- : eqtyp_=> eqtyp_ eqtyp_ eqtyp_.

eqtyp_all : ({w} eqtyp (A w) (A' w)) ->
            eqtyp (all A) (all A') -> type.
%mode eqtyp_all +A -B.
- : eqtyp_all ([w] eqtyp_) eqtyp_.

eqtyp_exists : ({w} eqtyp (A w) (A' w)) ->
               eqtyp (exists A) (exists A') -> type.
%mode eqtyp_exists +A -B.
- : eqtyp_exists ([w] eqtyp_) eqtyp_.

eqtyp_at : {W:world} eqtyp A A' -> eqtyp (A at W) (A' at W) -> type.
%mode eqtyp_at +W +A -B.
- : eqtyp_at _ eqtyp_ eqtyp_.

eqtyp_sh : ({w} eqtyp (A w) (A' w)) -> eqtyp (sh A) (sh A') -> type.
%mode eqtyp_sh +A -B.
- : eqtyp_sh ([_] eqtyp_) eqtyp_.

%worlds (blockw) (eqtyp_& _ _ _) (eqtyp_=> _ _ _) (eqtyp_all _ _)
                 (eqtyp_exists _ _) (eqtyp_at _ _ _) (eqtyp_sh _ _).
%total D (eqtyp_& D _ _).
%total D (eqtyp_=> D _ _).
%total D (eqtyp_all D _).
%total D (eqtyp_exists D _).
%total D (eqtyp_at _ D _).
%total D (eqtyp_sh D _).

of_resp : of M A W -> eqtyp A A' -> of M A' W -> type.
%mode of_resp +BOF +EQ -BOF'.
of_resp_ : of_resp D eqtyp_ D.

ofv_resp : ofv M A W -> eqtyp A A' -> ofv M A' W -> type.
%mode ofv_resp +BOF +EQ -BOF'.
ofv_resp_ : ofv_resp D eqtyp_ D.


ettoit_fun : ettoit A A' -> ettoit A A'' -> eqtyp A' A'' -> type.
%mode ettoit_fun +X +Y -Z.
```

```
- : ettoit_fun (ettoit_& E1 E2) (ettoit_& E1' E2') D3
 <- ettoit_fun E1 E1' D1
 <- ettoit_fun E2 E2' D2
 <- eqtyp_& D1 D2 D3.

- : ettoit_fun (ettoit_=> E1 E2) (ettoit_=> E1' E2') D3
 <- ettoit_fun E1 E1' D1
 <- ettoit_fun E2 E2' D2
 <- eqtyp_=> D1 D2 D3.

- : ettoit_fun (ettoit_at E) (ettoit_at E') D'
 <- ettoit_fun E E' D
 <- eqtyp_at _ D D'.

- : ettoit_fun (ettoit_sh E) (ettoit_sh E') D'
 <- ({w} ettoit_fun (E w) (E' w) (D w))
 <- eqtyp_sh D D'.

- : ettoit_fun ettoit_addr ettoit_addr eqtyp_.

- : ettoit_fun (ettoit_! E) (ettoit_! E') D'
 <- ettoit_fun E E' D
 <- eqtyp_=> (eqtyp_ : eqtyp unit unit) D Da
 <- ({w} eqtyp_at w Da (Dat w))
 <- eqtyp_all Dat D'.

- : ettoit_fun (ettoit_? E) (ettoit_? E') D'
 <- ettoit_fun E E' D
 <- ({w} eqtyp_at w D (Dat w))
 <- ({w} eqtyp_& (Dat w) eqtyp_ (Dand w))
 <- eqtyp_exists Dand D'.

%worlds (blockw) (ettoit_fun _ _ _).
%total D (ettoit_fun D _ _).

ettoit_gimme : {A:tye} {A':ty} ettoit A A' -> type.
%mode ettoit_gimme +A -A' -D.
ettoitf_gimme : {Af:world -> tye} {Af':world -> ty} ettoitf Af Af' -> type.
%mode ettoitf_gimme +Af -Af' -D.

- : ettoit_gimme (A &e B) _ (ettoit_& Da Db)
 <- ettoit_gimme A _ Da
 <- ettoit_gimme B _ Db.

- : ettoit_gimme (A ate W) _ (ettoit_at D)
 <- ettoit_gimme A _ D.

- : ettoit_gimme (A =>e B) _ (ettoit_=> Da Db)
 <- ettoit_gimme A _ Da
 <- ettoit_gimme B _ Db.

- : ettoit_gimme (!e A) _ (ettoit_! D)
 <- ettoit_gimme A _ D.

- : ettoit_gimme (?e A) _ (ettoit_? D)
 <- ettoit_gimme A _ D.

- : ettoit_gimme (addre _) _ ettoit_addr.

- : ettoit_gimme (she Af) _ (ettoit_sh D)
 <- ({w} ettoit_gimme (Af w) _ (D w)).

- : ettoitf_gimme Af Af' (ettoitf_ D)
 <- ({w} ettoit_gimme (Af w) (Af' w) (D w)).

%worlds (blockw) (ettoit_gimme _ _ _) (ettoitf_gimme _ _ _).
```

```
mobemob : mobilee A -> ettoit A A' -> mobile A' -> type.
%mode mobemob +BM +T -MOB.

bmm& : mobemob (&Me B1 B2) (ettoit_& T1 T2) (&M M1 M2)
    <- mobemob B1 T1 M1
    <- mobemob B2 T2 M2.

bmmat : mobemob atMe _ atM.
bmmaddr : mobemob addrMe _ addrM.
bmmsh : mobemob shaMe _ shaM.
bmm! : mobemob boxMe (ettoit_! _) (allM [w] atM).
bmm? : mobemob diaMe (ettoit_? _) (existsM [w] &M atM addrM).

%worlds (blockw) (mobemob _ _ _).
%total D (mobemob D _ _).

% converts typing derivations from EL to IL
elab+ : {A:tye} {E : expe} ofe E A W ->
        ettoit A A' ->
        {M : exp} of M A' W -> type.
%mode elab+ +A +B +BW +BT -E -OE.

elab- : {A:tye} {E : expe} ofe E A W ->
        ettoit A A' ->
        {M : exp} of M A' W -> type.
%mode elab- +A +B +BW -BT -E -OE.

elabv+ : {A:tye} {V : vale} ofve V A W ->
         ettoit A A' ->
         {VV : val} ofv VV A' W -> type.
%mode elabv+ +A +B +BW +BT -E -OE.

elabvv: {Af:world -> tye} {V : vvale} ofvve V Af ->
        ettoitf Af Af' ->
        {VV : vval} ofvv VV Af' -> type.
%mode elabvv +A +B +C -D -E -F.

elabv- : {A:tye} {V : vale} ofve V A W ->
         ettoit A A' ->
         {VV : val} ofv VV A' W -> type.
%mode elabv- +A +B +BW -BT -E -OE.

% need to convince Twelf that the I/O behavior
% of the ettoit argument is irrelevant, since
% it's a total function of the first argument.
el : elab+ A B BW BTi E OE'
  <- elab- A B BW BTo E OE
  <- ettoit_fun BTo BTi EQ
  <- of_resp OE EQ OE'.

% translate away the pair constructor
- : elab- _ _ (&Ie D1 D2) (ettoit_& ET1 ET2) _
   (oflet D1' [x][wx]
    oflet D2' [y][wy]
      ofvalue (&Iv wx wy))
  <- elab- _ _ D1 ET1 _ D1'
  <- elab- _ _ D2 ET2 _ D2'.

- : elab- _ _ (&E1e D) ET _ (&E1 D')
 <- elab- _ _ D (ettoit_& ET _) _ D'.
- : elab- _ _ (&E2e D) ET _ (&E2 D')
 <- elab- _ _ D (ettoit_& _ ET) _ D'.

- : elab- _ _ oflocalhoste ettoit_addr _ oflocalhost.
```

```
- : elab- _ _ (=>Ee Df Da) ET _ (=>E Df' Da')
 <- elab- _ _ Df (ettoit_=> ET' ET) _ Df'
 <- elab+ _ _ Da ET' _ Da'.

- : elab- _ _ (atEe Dm Dn) ET _ (atE Dm' Dn')
 <- ettoit_gimme A A' ETv
 <- elab+ _ _ Dm (ettoit_at ETv) _ Dm'
 <- ({ve : vale}{ove : ofve ve A W}
     {v  : val} {ov  : ofv  v  A' W} elabv- A ve ove ETv v ov ->
       elab- _ _ (Dn ve ove) ET _ (Dn' v ov)).

- : elab- _ _ (ofpute (ME : mobilee A) (Dm : ofe M A W)
                (Dn : ({u:vvale} ofvve u ([w] A) ->
                        ofe (N u) C W))
              : ofe _ C W) (ET : ettoit C C') _ (ofput MI Dm' Dn')
 <- ettoit_gimme A A' ETa
 <- elab+ _ _ Dm ETa _ Dm'
 <- mobemob ME ETa MI
 <- ({ve : vvale}{ove : ofvve ve ([w] A)}
     {v  : vval} {ov  : ofvv  v  ([w] A')}
     elabvv ([w] A) ve ove (ettoitf_ [w] ETa) v ov ->
       elab- _ _ (Dn ve ove) ET _ (Dn' v ov)).

% interesting cases
- : elab- _ _ (!Ee D) ET _
         (atE (allE D') [x][ofx] =>E (ofvalue ofx) (ofvalue unitI))
 <- elab- _ _ D (ettoit_! ET) _ D'.

- : elab- _ _ (?Ie D) (ettoit_? ET) _
          (oflet D' [x][ofx]
           oflet oflocalhost [a][ofa]
           oflet (ofvalue (&Iv (atIv ofx) ofa)) [p][ofp]
             ofvalue (existsI _ ofp))
 <- elab- _ _ D ET _ D'.

- : elab- _ _ (ofgete ME Da Dm) ET _ (ofget ME' Da' Dm')
 <- elab+ _ _ Da ettoit_addr _ Da'
 <- elab- _ _ Dm ET _ Dm'
 <- mobemob ME ET ME'.

- : elab- _ _ (?Ee Dm Dn) ET _
     (existsE Dm' [w][p][ofp]
        oflet (&E1 (ofvalue ofp)) [v][ov]
        oflet (&E2 (ofvalue ofp)) [a][oa]
        atE (ofvalue ov) [v'][ov']
        Dn' w a oa v' ov')
 <- ettoit_gimme C C' ET
 <- elab- _ _ Dm (ettoit_? ETv) _ Dm'
 <- ({w : world}
     % address
     {ae : vale}{oae : ofve ae (addre w) W}
     {a  : val} {oa  : ofv  a  (addr w) W}
     {_ : elabv- (addre w) ae oae ettoit_addr a oa}
     % value
     {ve : vale}{ove : ofve ve A w}
     {v  : val} {ov  : ofv  v  A' w}
     {_ : elabv- A ve ove ETv v ov}
       elab+ _ _ (Dn w ae oae ve ove) ET _ (Dn' w a oa v ov)).

- : elab- _ _ (shEe Dm Dn) ET _ (shE Dm' Dn')
 <- ettoitf_gimme A A' (ettoitf_ ETv)
 <- elab+ _ _ Dm (ettoit_sh ETv) _ Dm'
 <- ({ve : vvale}{ove : ofvve ve A}
     {v : vval}{ov : ofvv v A'}
     {thm : elabvv A ve ove (ettoitf_ ETv) v ov}
       elab- _ _ (Dn ve ove) ET _ (Dn' v ov)).
```

```
% we don't need 'hold' in the IL because we can just
% sequence the evaluation and use held.
- : elab- _ _ (atIe D) (ettoit_at ET) (let M' ([y] value (held y)))
          (oflet D' ([y:val][wy:ofv y A' W] ofvalue (atIv wy)))
 <- elab- _ _ D ET _ (D' : of M' A' W).

% other judgments
- : elab- _ _ (ofvaluee D) ET _ (ofvalue D') <- elabv- _ _ D ET _ D'.

% elaboration of value judgment

elv : elabv+  A B BW BTi E OE'
   <- elabv- A B BW BTo E OE
   <- ettoit_fun BTo BTi EQ
   <- ofv_resp OE EQ OE'.

- : elabv- _ _ (addrIe : ofve _ _ W) ettoit_addr _
               (addrI : ofv _ _ W).

- : elabv- _ _ (&Ive Da Db) (ettoit_& ETa ETb) _ (&Iv Da' Db')
 <- elabv- _ _ Da ETa _ Da'
 <- elabv- _ _ Db ETb _ Db'.

- : elabv- _ _ (=>Ie D) (ettoit_=> ETa ETb) _ (=>I D')
 <- ettoit_gimme A A' ETa
 <- ({ve : vale}{ove : ofve ve A W}
     {v  : val} {ov  : ofv  v  A' W}
     {thm : elabv- A ve ove ETa v ov}
       elab- _ _ (D ve ove) ETb _ (D' v ov)).

- : elabv- _ _ (atIve D) (ettoit_at ET) _ (atIv D')
 <- elabv- _ _ D ET _ D'.

- : elabv- _ _ (shIe Dv) (ettoit_sh ET) _ (shI Dv')
 <- elabvv _ _ Dv (ettoitf_ ET) _ Dv'.

- : elabv- _ _ (ofvvalide (WVV : ofvve VV Af))
               (ETX W : ettoit (Af W) (Af' W))  _ (ofvvalid WVV')
 <- elabvv _ _ WVV ((ettoitf_ ETX) : ettoitf Af Af')  _ WVV'.

- : elabvv Af _ (vvIe (WV : {w} ofve (Vf w) (Af w) w))
                (ettoitf_ CTf) _ (vvI (WV' : {w} ofv (Vf' w) (Af' w) w))
    <- ettoitf_gimme Af Af' (ettoitf_ CTf)
    <- ({w} elabv+ _ _ (WV w) (CTf w) _  (WV' w)).

% interesting cases
- : elabv- (?e A) (theree Va V) (?Ive (WVa : ofve Va (addre W') W) WV)
     (ettoit_? ET)
     _ (existsI _ (&Iv (atIv WV') (WVa')))
 <- ettoit_gimme _ _ ET
 <- elabv+ (addre W') Va WVa ettoit_addr _ WVa'
 <- elabv+ _ _ WV ET _ WV'.

- : elabv- _ _ (!Ie D) (ettoit_! ET) _
     (allI [w] atIv (=>I [u][ofu] (D' w)))
 <- ettoit_gimme _ _ ET
 <- ({w : world} elab+ _ _ (D w) ET _ (D' w)).

%block blockve :
  some  {A:tye}{A':ty}{W:world}{ETv : ettoit A A'}
  block {ve : vale}{ove : ofve ve A W}{v : val}{ov : ofv v A' W}
        {thm : elabv- A ve ove ETv v ov}.

%block blockvve :
  some  {Af:world -> tye}{Af':world -> ty}{ETf : ettoitf Af Af'}
```

```
  block {ve : vvale}{ove : ofvve ve Af}{v : vval}{ov : ofvv v Af'}
        {thm : elabvv Af ve ove ETf v ov}.

%worlds (blockw | blockve | blockvve)
  (elab+ _ _ _ _ _ _) (elab- _ _ _ _ _ _)
  (elabv+ _ _ _ _ _ _) (elabv- _ _ _ _ _ _)
  (elabvv _ _ _ _ _ _) (ofv_resp _ _ _) (of_resp _ _ _).

%total (D E) (ettoit_gimme D _ _) (ettoitf_gimme E _ _).
%total D (of_resp _ D _).
%total D (ofv_resp _ D _).

%total (C B A Z Y)  (elab- _ _ C _ _ _) (elab+ _ _ B _ _ _)
  (elabv- _ _ A _ _ _) (elabv+ _ _ Z _ _ _) (elabvv _ _ Y _ _ _).
```

A.8.6 CPS conversion and static correctness

```
% CPS Conversion.                                2 Oct 2006, 24 Oct 2007

% Again, given on typing derivations. This builds the type-correctness
% of the translation into the argument.


% convert an IL type to a CPS type.
% ttoct : typ -> ctyp -> type.                      %name ttoct TTOCT ttoct.
%mode ttoct +T -CT.
% ttoctf : (world -> typ) -> (world -> ctyp) -> type.
%mode ttoctf +T -CT.

ttoct/at : ttoct (A at W) (A' cat W) <- ttoct A A'.
ttoct/=> : ttoct (A => B) ((A' c& (B' ccont)) ccont) <- ttoct B B' <- ttoct A A'.
ttoct/all : ttoct (all A) (call A') <- ({w} ttoct (A w) (A' w)).
ttoct/exists : ttoct (exists A) (cexists A') <- ({w} ttoct (A w) (A' w)).
ttoct/addr : ttoct (addr W) (caddr W).
ttoct/unit : ttoct unit cunit.
ttoct/sh : ttoct (sh A) (csh A') <- ({w} ttoct (A w) (A' w)).
ttoct/& : ttoct (A & B) (A' c& B') <- ttoct B B' <- ttoct A A'.

ttoctf/ : ttoctf Af Af' <- ({w : world} ttoct (Af w) (Af' w)).

% sanity check
%worlds (blockw) (ttoct _ _) (ttoctf _ _).
%total (D E) (ttoct D _) (ttoctf E _).

% type translation preserves mobility.
cmobmob : ttoct A A' -> mobile A -> cmobile A' -> type.
%mode cmobmob +T +M -CM.

- : cmobmob _ atM cmob_at.
- : cmobmob _ addrM cmob_addr.
- : cmobmob _ shaM cmob_sh.
- : cmobmob (ttoct/& A B) (&M AM BM) (cmob_& ACM BCM)
 <- cmobmob A AM ACM
 <- cmobmob B BM BCM.
- : cmobmob (ttoct/all A) (allM MOB) (cmob_all MOBC)
 <- ({w} cmobmob (A w) (MOB w) (MOBC w)).
- : cmobmob (ttoct/exists A) (existsM MOB) (cmob_exists MOBC)
 <- ({w} cmobmob (A w) (MOB w) (MOBC w)).


% it has to have unique otputs.
ttoct_fun : ttoct A A' -> ttoct A A'' -> ceqtyp A' A'' -> type.
%mode ttoct_fun +X +Y -Z.
```

```
- : ttoct_fun (ttoct/& A B) (ttoct/& C D) OUT
 <- ttoct_fun A C EQ1
 <- ttoct_fun B D EQ2
 <- ceqtyp_& EQ1 EQ2 OUT.

- : ttoct_fun (ttoct/=> A B) (ttoct/=> C D) OUT
 <- ttoct_fun A C EQ1
 <- ttoct_fun B D EQ2
 <- ceqtyp_cont EQ2 EQ3
 <- ceqtyp_& EQ1 EQ3 EQ4
 <- ceqtyp_cont EQ4 OUT.

- : ttoct_fun ttoct/addr ttoct/addr ceqtyp_.
- : ttoct_fun ttoct/unit ttoct/unit ceqtyp_.

- : ttoct_fun (ttoct/at A) (ttoct/at B) OUT
 <- ttoct_fun A B EQ
 <- ceqtyp_at W EQ OUT.

- : ttoct_fun (ttoct/all A) (ttoct/all B) OUT
 <- ({w} ttoct_fun (A w) (B w) (EQ w))
 <- ceqtyp_all EQ OUT.

- : ttoct_fun (ttoct/exists A) (ttoct/exists B) OUT
 <- ({w} ttoct_fun (A w) (B w) (EQ w))
 <- ceqtyp_exists EQ OUT.

- : ttoct_fun (ttoct/sh A) (ttoct/sh B) OUT
 <- ({w} ttoct_fun (A w) (B w) (EQ w))
 <- ceqtyp_sh EQ OUT.

% moreover, it must be defined for all inputs
ttoct_gimme  : {A:ty} {A':ctyp} ttoct A A' -> type.
%mode ttoct_gimme +A -A' -D.
ttoctf_gimme : {Af:world -> ty} {Af':world -> ctyp} ttoctf Af Af' -> type.
%mode ttoctf_gimme +A -A' -D.

- : ttoct_gimme (A & B) _ (ttoct/& CT1 CT2)
 <- ttoct_gimme A A' CT1
 <- ttoct_gimme B B' CT2.

- : ttoct_gimme (A => B) _ (ttoct/=> CT1 CT2)
 <- ttoct_gimme A A' CT1
 <- ttoct_gimme B B' CT2.

- : ttoct_gimme (A at W) _ (ttoct/at CT)
 <- ttoct_gimme A A' CT.

- : ttoct_gimme (addr W) _ ttoct/addr.
- : ttoct_gimme unit cunit ttoct/unit.

- : ttoct_gimme (all A) _ (ttoct/all CT)
 <- ({w} ttoct_gimme (A w) (A' w) (CT w)).

- : ttoct_gimme (exists A) _ (ttoct/exists CT)
 <- ({w} ttoct_gimme (A w) (A' w) (CT w)).

- : ttoct_gimme (sh Af) (csh Af') (ttoct/sh CT)
 <- ({w} ttoct_gimme (Af w) (Af' w) (CT w)).

- : ttoctf_gimme Af Af' (ttoctf/ D)
 <- ({w} ttoct_gimme (Af w) (Af' w) (D w)).

ttoct_gimme_at : {W:world} {A:ty} {A':ctyp} ttoct (A at W) (A' cat W) -> type.
%mode ttoct_gimme_at +W +A -A' -D.
```

```
- : ttoct_gimme_at W A A' (ttoct/at D) <- ttoct_gimme A A' D.

ttoct_gimme_sh : {A:world -> ty} {A':world -> ctyp} ttoct (sh A) (csh A') -> type.
%mode ttoct_gimme_sh +A -A' -D.

- : ttoct_gimme_sh A A' (ttoct/sh D) <- ({w} ttoct_gimme (A w) (A' w) (D w)).

%worlds (blockw) (ttoct_fun _ _ _) (ceqtyp_& _ _ _) (ceqtyp_cont _ _) (ceqtyp_at _ _ _)
                 (ceqtyp_all _ _) (ceqtyp_exists _ _) (ceqtyp_sh _ _).

%total D (ceqtyp_& D _ _).
%total D (ceqtyp_cont D _).
%total D (ceqtyp_sh D _).
%total D (ceqtyp_at D _ _).
%total D (ceqtyp_all D _).
%total D (ceqtyp_exists D _).
%total D (ttoct_fun D _ _).

tocps- : {M  : exp}
         {WM : of M A W}
         {CT : ttoct A CA}
         % this term represents the result of conversion.
         % it takes a continuation with a hole for the
         % translated result and fills it in.
         {CC : (cval -> cexp) -> cexp}
         % this derivation types the result of conversion:
         % All well-formed instantiations of CC are well-formed.
         % For C to be a well-formed instantiation, it must
         % itself be well-typed for all appropriate arguments.
         ({C : cval -> cexp}
          ({cv : cval}
           {wcv : cofv cv CA W}
           cof (C cv) W) ->
          % result of course can depend on C
          cof (CC C) W) ->
         type.
%mode tocps- +M +WM -CT -CC -WCC.

% need this because Twelf doesn't understand that ttoct is
% a function. Reverse the polarity!
tocps+ : {M  : exp}
         {WM : of M A W}
         {CT : ttoct A CA}
         {CC : (cval -> cexp) -> cexp}
         ({C : cval -> cexp}
          ({cv : cval}
           {wcv : cofv cv CA W}
           cof (C cv) W) ->
          cof (CC C) W) ->
         type.
%mode tocps+ +M +WM +CT -CC -WCC.

% well-formedness of continuation must respect equality on
% CPS types, for reverse-the-polarity trick.
wcc_resp : {WCC :
    ({C  : cval -> cexp}
     ({cv : cval}
      {wcv : cofv cv A W}
      cof (C cv) W) ->
     cof (CC C) W)}
   {EQ : ceqtyp A A'}
   {WCC' :
    ({C  : cval -> cexp}
     ({cv : cval}
      {wcv : cofv cv A' W}
      cof (C cv) W) ->
```

```
     cof (CC C) W)}
   type.
%mode wcc_resp +WCC +EQ -WCC'.
wcc_resp_ : wcc_resp D ceqtyp_ D.


tocps+/- : tocps+ V WV CTi CC WCC
        <- tocps- V WV CTo CC WCC'
        <- ttoct_fun CTo CTi EQ
        <- wcc_resp WCC' EQ WCC.

% values are translated with a more standard-looking relation
tocpsv+ : {WV  : ofv V A W}
          {CT  : ttoct A CA}
          {WCV : cofv CV CA W}
          type.
%mode tocpsv+ +WV +CT -WCV.
%mode tocpsv- +WV -CT -WCV.

% forward declared
% tocpsvv- : {WV : ofvv V A}
%            {CT : ttoctf A CA}
%            {WCV : cofvv CV CA}
%            type.
%mode tocpsvv- +WV -CT -WCV.

tocpsv+/- : tocpsv+ WV CTi WCV'
         <- tocpsv- WV CTo WCV
         <- ttoct_fun CTo CTi EQ
         <- cofv_resp WCV EQ WCV'.

c_val : tocps- (value V) (ofvalue WV) CT _ ([c][wc] wc CV WCV)
     <- tocpsv- WV CT WCV.

c_fst : tocps- (fst M) (&E1 WM) CT _
             ([c : cval -> cexp][wc : ({cv : cval}
{wcv : cofv cv CA W}
  cof (c cv) W)]
                F _ ([v][wv] co_fst wv wc))
     <- tocps- M WM (ttoct/& CT _) _ F.

c_snd : tocps- (snd M) (&E2 WM) CT _
             ([c][wc]
                F _ ([v][wv] co_snd wv wc))
     <- tocps- M WM (ttoct/& _ CT) _ F.

c_localhost : tocps- _ oflocalhost ttoct/addr _ ([c][wc] co_localhost wc).

c_unpack : tocps- (unpack M N) (existsE WM WN) CTN _
              ([c][wc]
                 FM _ ([v][wv]
                         co_unpack wv ([w][x][xof]
                                         FN w x xof c wc)))
        <- ttoct_gimme (exists A) (cexists A') (ttoct/exists CTM)
        <- tocps+ M WM (ttoct/exists CTM) _ FM
        <- ttoct_gimme B B' CTN
        <- ( {w}
             {x}{xof : ofv x (A w) W}
             {x'}{x'of : cofv x' (A' w) W}
             {thm: tocpsv- xof (CTM w) x'of}

            tocps+ (N w x) (WN w x xof) CTN (CC w x') (FN w x' x'of)).

c_app : tocps- (app M N) (=>E WM WN) CTB _
           ([c][wc]
              % eval function, then argument
```

```
             FM _ ([f][wf]
                    FN _ ([a][wa]
                            co_call wf (cov_pair wa (cov_lam ([r][wr] wc r wr)) ))))
     <- ttoct_gimme (A => B) (A' c& (B' ccont) ccont) (ttoct/=> CTA CTB)
     <- tocps+ M WM (ttoct/=> CTA CTB) _ FM
     <- tocps+ N WN CTA _ FN.

c_wapp : tocps- (wapp M W') (allE WM) (CTM W') _
            ([c][wc]
               FM _ ([v][wv]
                       co_wapp wv wc))
      <- ttoct_gimme (all A) (call A') (ttoct/all CTM)
      <- tocps+ M WM (ttoct/all CTM) _ FM.

c_leta : tocps- (leta M N) (atE (WM : of M (A at W') W) WN) CTN _
            ([c][wc]
               FM _ ([v][wv]
                       co_leta wv ([x][xof] FN x xof c wc)))
      <- ttoct_gimme_at W' A A' (ttoct/at CTA)
      <- tocps+ M WM (ttoct/at CTA) _ FM
      <- ttoct_gimme B B' CTN
      <- ( {x}{xof : ofv x A _}
           {x'}{x'of : cofv x' A' _}
           {thm:tocpsv- xof CTA x'of}

          tocps+ (N x) (WN x xof) CTN (CC x') (FN x' x'of)).

% we just translate away lets
c_let : tocps- (let M N) (oflet WM WN) CTN _
          ([c][wc] FM _ ([v][wv] FN v wv c wc))

     <- ttoct_gimme A A' CTM
     <- tocps+ M WM CTM _ FM
     <- ttoct_gimme B B' CTN
     <- ( {x}{xof : ofv x A _}
          {x'}{x'of : cofv x' A' _}
          {thm:tocpsv- xof CTM x'of}

         tocps+ (N x) (WN x xof) CTN (CC x') (FN x' x'of)).

% interesting
c_get : tocps- (get W MA M) (ofget MOB WMA WM) CT _
         ([c][wc]
  % eval remote addr
  FMA _ ([a][wa]
 % get our addr
 co_localhost [h][wh]
 % make it valid
 co_put cmob_addr wh [uh][wuh]
 % head on over...
 co_go wa
   (FM _ ([m][wm]
  % make result valid
  co_put CMOB wm [um][wum]
  % and go back...
  co_go (cov_valid wuh) (wc (cvalid um) (cov_valid wum))))))
     <- ttoct_gimme A A' CT
     <- tocps+ MA WMA ttoct/addr _ FMA
     <- tocps+ M WM CT _ FM
     <- cmobmob CT MOB CMOB.

c_lets : tocps- (lets M N) (shE (WM : of M (sh Af) W) WN) CTN _
            ([c][wc]
               FM _ ([v][wv]
                       co_lets wv ([x][xof] FN x xof c wc)))
      <- ttoct_gimme_sh Af Af' (ttoct/sh CTA)
```

```
      <- tocps+ M WM (ttoct/sh CTA) _ FM
      <- ttoct_gimme B B' CTN
      <- ( {x}{xof : ofvv x Af}
           {x'}{x'of : cofvv x' Af'}
           {thm:tocpsvv- xof (ttoctf/ CTA) x'of}

          tocps+ (N x) (WN x xof) CTN (CC x') (FN x' x'of)).

c_put : tocps- (put M N) (ofput (MOB : mobile A) (WM : of M A W) WN) CTN _
            ([c][wc]
               FM _ ([v][wv]
                       co_put CMOB wv ([x][xof] FN x xof c wc)))
      <- ttoct_gimme A A' CTA
      <- tocps+ M WM CTA _ FM
      <- ttoct_gimme B B' CTN
      <- cmobmob CTA MOB CMOB
      <- ( {x}{xof : ofvv x ([w] A)}
           {x'}{x'of : cofvv x' ([w] A')}
           {thm:tocpsvv- xof (ttoctf/ [w] CTA) x'of}

          tocps+ (N x) (WN x xof) CTN (CC x') (FN x' x'of)).

cv_pair : tocpsv- (&Iv WV1 WV2) (ttoct/& CT1 CT2)  (cov_pair WV1' WV2')
       <- tocpsv- WV1 CT1 WV1'
       <- tocpsv- WV2 CT2 WV2'.

cv_held : tocpsv- (atIv WV) (ttoct/at CT)  (cov_held WV')
           <- tocpsv- WV CT WV'.

% this is essentially the crux of continuation *passing* style
cv_lam : tocpsv- ((=>I WM) : ofv (lam M) (A => B) W) (ttoct/=> CTA CTB)
          (cov_lam [arg][argof : cofv arg (A' c& (B' ccont)) W]
             co_fst argof [x:cval][xof:cofv x A' W]
             co_snd argof [r:cval][rof:cofv r (B' ccont) W]
             F x xof r rof ([v:cval] ccall r v)
                ([v:cval][wv:cofv v B' W] co_call rof wv)
             )
    <- ttoct_gimme A A' CTA
    <- ttoct_gimme B B' CTB
    <- (% original argument
        {x}{xof : ofv x A W}
        {x'}{x'of : cofv x' A' W}
        % how to convert it
        {thm:tocpsv- xof CTA x'of}

        % (object language) return continuation
        {r}{rof : cofv r (B' ccont) W}

          tocps+ (M x) (WM x xof) CTB (CC x' r) (F x' x'of r rof)).

cv_addr : tocpsv- addrI ttoct/addr cov_const.

cv_all : tocpsv- (allI WV) (ttoct/all CT)  (cov_wlam V')
          <- ({w:world} tocpsv- (WV w) (CT w) (V' w)).

cv_unit : tocpsv- unitI ttoct/unit cov_unit.

cv_pack : tocpsv- (existsI A WV) (ttoct/exists CT)  (cov_pack A' WV')
          <- ({w} ttoct_gimme (A w) (A' w) (CT w))
          <- (tocpsv+ WV (CT W') WV').

cv_vvalid : tocpsv- (ofvvalid (WVV : ofvv VV Af))
                    (CTX W : ttoct (Af W) (Af' W))  (cov_valid WVV')
         <- tocpsvv- WVV ((ttoctf/ CTX) : ttoctf Af Af')  WVV'.

cv_shI : tocpsv- (shI WVV)  (ttoct/sh CTA) (cov_ch WVV')
```

```
      <- tocpsvv- WVV (ttoctf/ CTA) WVV'.

cvv_ : tocpsvv- (vvI (WV : {w} ofv (Vf w) (Af w) w))
                (ttoctf/ CTf) (covv (WV' : {w} cofv (Vf' w) (Af' w) w))
    <- ttoctf_gimme Af Af' (ttoctf/ CTf)
    <- ({w} tocpsv+ (WV w) (CTf w)  (WV' w)).

%worlds (blockw | blockcvar | blockwcv | blockcvvar)
  (tocps+ _ _ _ _ _) (tocps- _ _ _ _ _) (tocpsv+ _ _ _) (tocpsv- _ _ _)
  (tocpsvv- _ _ _) (cofv_resp _ _ _) (ttoct_gimme _ _ _)
  (ttoctf_gimme _ _ _) (ttoct_gimme_at _ _ _ _) (ttoct_gimme_sh _ _ _)
  (cmobmob _ _ _) (wcc_resp _ _ _).

%total D (cmobmob _ D _).
%total A (cofv_resp _ A _).
%total A (wcc_resp _ A _).
%total A (ttoct_gimme A _ _).
%total A (ttoctf_gimme A _ _).
%total A (ttoct_gimme_at _ A _ _).
%total A (ttoct_gimme_sh A _ _).

%total (A B C D E) (tocpsv- A _ _) (tocpsv+ B _ _) (tocps- _ C _ _ _)
                   (tocps+ _ D _ _ _) (tocpsvv- E _ _).
```

A.8.7 Closure conversion and static correctness

```
% Closure conversion

% the type syntax is the same as for CPS.
ccexp : type.    %name ccexp CC cc.
ccval : type.    %name ccval V v.
ccvval : type.   %name ccvval VV vv.

% continuation expressions
ccfst : ccval -> (ccval -> ccexp) -> ccexp.
ccsnd : ccval -> (ccval -> ccexp) -> ccexp.
cclocalhost : (ccval -> ccexp) -> ccexp.
cclets : ccval -> (ccvval -> ccexp) -> ccexp.
ccput : ccval -> (ccvval -> ccexp) -> ccexp.
ccleta : ccval -> (ccval -> ccexp) -> ccexp.
ccunpack : ccval -> (world -> ccval -> ccexp) -> ccexp.
ccwapp : ccval -> world -> (ccval -> ccexp) -> ccexp.

% the closure-converted language has a different 'go'
% which takes a closure value to execute at the remote world
ccgo : world -> ccval -> ccval -> ccexp.
cccall : ccval -> ccval -> ccexp.
cchalt : ccexp.

% continuation values
ccpair : ccval -> ccval -> ccval.
ccheld : ccval -> ccval.
%       argument -> environment -> body    env
ccclosure : (ccval -> ccval -> ccexp) -> ccval -> ccval.
ccconst : world -> ccval.
ccwlam : (world -> ccval) -> ccval.
ccpack : world -> ccval -> ccval.
cc1 : ccval.
ccvalid : ccvval -> ccval.
ccsh : ccvval -> ccval.
ccvv : (world -> ccval) -> ccvval.

%block blockccv : block {x:ccval}.
%block blockccvv : block {xx:ccvval}.
```

```
%worlds (blockw | blockccv | blockccvv) (ccval) (ccvval) (ccexp).

% a variable is frozen in an exp/val/vval if it does not
% appear within any closure bodies.
frozen     : (ccval  -> ccexp)  -> type.        %name frozen Z z.
vfrozen    : (ccval  -> ccval)  -> type.        %name vfrozen Z z.
vvfrozen   : (ccval  -> ccvval) -> type.        %name vvfrozen Z z.

frozenvv   : (ccvval -> ccexp)  -> type.        %name frozenvv ZZ zz.
vfrozenvv  : (ccvval -> ccval)  -> type.        %name vfrozenvv ZZ zz.
vvfrozenvv : (ccvval -> ccvval) -> type.        %name vvfrozenvv ZZ zz.

f/closed : frozen ([x] M).
f/fst : frozen ([x] ccfst (V x) ([y] C y x))
     <- ({y} frozen (C y))
     <- vfrozen V.
f/snd : frozen ([x] ccsnd (V x) ([y] C y x))
     <- ({y} frozen (C y))
     <- vfrozen V.
f/localhost : frozen ([x] cclocalhost ([u] C u x)) <- ({y} frozen (C y)).
f/lets : frozen ([x] cclets (V x) ([u] C u x))
          <- ({u} frozen (C u))
          <- vfrozen V.
f/put : frozen ([x] ccput (V x) ([u] C u x))
      <- ({u} frozen (C u))
      <- vfrozen V.
f/leta : frozen ([x] ccleta (V x) ([y] C y x))
      <- ({y} frozen (C y))
      <- vfrozen V.
f/unpack : frozen ([x] ccunpack (V x) ([w][y] C w y x))
        <- ({w:world}{y:ccval} frozen (C w y))
        <- vfrozen V.
f/wapp : frozen ([x] ccwapp (V x) W ([y] C y x))
      <- ({y} frozen (C y))
      <- vfrozen V.
f/go : frozen ([x] ccgo W (V1 x) (V2 x))
    <- vfrozen V2
    <- vfrozen V1.
f/call : frozen ([x] cccall (V1 x) (V2 x))
      <- vfrozen V2
      <- vfrozen V1.
f/halt : frozen ([x] cchalt).
vf/valid : vfrozen ([x] ccvalid (VV x)) <- vvfrozen VV.
vf/pair : vfrozen ([x] ccpair (V1 x) (V2 x))
       <- vfrozen V2
       <- vfrozen V1.
vf/held : vfrozen ([x] ccheld (V x)) <- vfrozen V.
vf/const : vfrozen ([x] ccconst W).
vf/1 : vfrozen ([x] cc1).
vf/wlam : vfrozen ([x] ccwlam ([w] V w x)) <- ({w} vfrozen (V w)).
vf/pack : vfrozen ([x] ccpack W (V x)) <- vfrozen V.
vf/var : vfrozen ([x] x).
vf/closed : vfrozen ([x] V).

%% crux: frozen variables cannot appear within the body of a closure.
vf/closure : vfrozen ([x] ccclosure ([a][e] BOD a e) (ENV x))
          <- vfrozen ENV.

vf/ch : vfrozen ([x] ccsh (VV x)) <- vvfrozen VV.
vvf/vv : vvfrozen ([x] ccvv ([w] V w x)) <- ({w} vfrozen (V w)).
vvf/closed : vvfrozen ([x] VV).

%% same as above, but for valid vars.
fvv/fst : frozenvv ([x] ccfst (V x) ([y] C y x))
     <- ({y} frozenvv (C y))
     <- vfrozenvv V.
```

```
fvv/snd : frozenvv ([x] ccsnd (V x) ([y] C y x))
     <- ({y} frozenvv (C y))
     <- vfrozenvv V.
fvv/localhost : frozenvv ([x] cclocalhost ([u] C u x))
          <- ({y} frozenvv (C y)).
fvv/lets : frozenvv ([x] cclets (V x) ([u] C u x))
          <- ({u} frozenvv (C u))
          <- vfrozenvv V.
fvv/put : frozenvv ([x] ccput (V x) ([u] C u x))
      <- ({u} frozenvv (C u))
      <- vfrozenvv V.
fvv/leta : frozenvv ([x] ccleta (V x) ([y] C y x))
      <- ({y} frozenvv (C y))
      <- vfrozenvv V.
fvv/unpack : frozenvv ([x] ccunpack (V x) ([w][y] C w y x))
        <- ({w:world}{y:ccval} frozenvv (C w y))
        <- vfrozenvv V.
fvv/wapp : frozenvv ([x] ccwapp (V x) W ([y] C y x))
      <- ({y} frozenvv (C y))
      <- vfrozenvv V.
fvv/go : frozenvv ([x] ccgo W (V1 x) (V2 x))
      <- vfrozenvv V2
      <- vfrozenvv V1.
fvv/call : frozenvv ([x] cccall (V1 x) (V2 x))
      <- vfrozenvv V2
      <- vfrozenvv V1.
fvv/halt : frozenvv ([x] cchalt).
vfvv/valid : vfrozenvv ([x] ccvalid (VV x))
        <- vvfrozenvv VV.
vfvv/pair : vfrozenvv ([x] ccpair (V1 x) (V2 x))
       <- vfrozenvv V2
       <- vfrozenvv V1.
vfvv/held : vfrozenvv ([x] ccheld (V x)) <- vfrozenvv V.
vfvv/const : vfrozenvv ([x] ccconst W).
vfvv/1 : vfrozenvv ([x] cc1).
vfvv/wlam : vfrozenvv ([x] ccwlam ([w] V w x)) <- ({w} vfrozenvv (V w)).
vfvv/pack : vfrozenvv ([x] ccpack W (V x)) <- vfrozenvv V.
vfvv/closed : vfrozenvv ([u] V).
vfvv/closure : vfrozenvv ([x] ccclosure ([a][e] BOD a e) (ENV x))
          <- vfrozenvv ENV.
vfvv/ch : vfrozenvv ([x] ccsh (VV x)) <- vvfrozenvv VV.
vvfvv/vv : vvfrozenvv ([x] ccvv ([w] V w x)) <- ({w} vfrozenvv (V w)).
vvfvv/var : vvfrozenvv ([u] u).
vvfvv/closed : vvfrozenvv ([u] VV).
%% end duplicated

ccof : ccexp -> world -> type.               %name ccof WC wc.
ccofv : ccval -> ctyp -> world -> type.      %name ccofv WV wv.
ccofvv : ccvval -> (world -> ctyp) -> type.  %name ccofvv WVV wvv.

cco_halt : ccof cchalt W.

% different now: needs a closed continuation value
cco_go : ccofv VW (caddr W') W ->
         ccofv VC (cunit ccont) W' ->
         ccof (ccgo W' VW VC) W.

cco_put : cmobile A ->
           ccofv V A W ->
           ({v} ccofvv v ([w] A) ->
              ccof (N v) W) ->
           frozenvv N ->
           ccof (ccput V N) W.

cco_lets : ccofv V (csh Af) W ->
               ({v} ccofvv v Af ->
```

```
                ccof (N v) W) ->
              frozenvv N ->
              ccof (cclets V N) W.

cco_leta : ccofv V (A cat W') W ->
           ({v} ccofv v A W' ->
             ccof (N v) W) ->
           frozen N ->
           ccof (ccleta V N) W.

cco_wapp : ccofv V (call A) W ->
           ({v} ccofv v (A W') W ->
              ccof (N v) W) ->
           frozen N ->
           ccof (ccwapp V W' N) W.

cco_unpack :
          ccofv V (cexists A) W ->
          ({w}{v} ccofv v (A w) W ->
              ccof (N w v) W) ->
          ({w} frozen (N w)) ->
          ccof (ccunpack V N) W.

cco_localhost : ({v} ccofv v (caddr W) W ->
                     ccof (N v) W) ->
              frozen N ->
              ccof (cclocalhost N) W.

cco_call : ccofv F (A ccont) W ->
           ccofv V A W ->
           ccof (cccall F V) W.

cco_fst : ccofv V (A c& B) W ->
          ({v}{ov : ccofv v A W} ccof (C v) W) ->
          frozen C ->
          ccof (ccfst V C) W.

cco_snd : ccofv V (A c& B) W ->
          ({v}{ov : ccofv v B W} ccof (C v) W) ->
          frozen C ->
          ccof (ccsnd V C) W.

% this is new!
ccov_closure : ccofv ENV ENVT W ->
               ({x}{xof : ccofv x A W}
                {e}{eof : ccofv e ENVT W}
                ccof (BOD x e) W) ->
               ({x} frozen ([y] BOD x y)) ->
               ({y} frozen ([x] BOD x y)) ->
               ccofv (ccclosure BOD ENV) (A ccont) W.

ccov_pair : ccofv V1 A W ->
            ccofv V2 B W ->
            ccofv (ccpair V1 V2) (A c& B) W.

ccov_unit : ccofv cc1 cunit W.
ccov_held : ccofv V A W' -> ccofv (ccheld V) (A cat W') W.
ccov_const : ccofv (ccconst W) (caddr W) W'.

ccov_wlam : ({w} ccofv (V w) (A w) W) ->
            ccofv (ccwlam V) (call A) W.

ccov_pack : {A : world -> ctyp} % annotation
            ccofv V (A W') W ->
            ccofv (ccpack W' V) (cexists A) W.
```

```
ccov_valid : ccofvv VV Af -> ccofv (ccvalid VV) (Af W) W.
ccov_ch : ccofvv VV A -> ccofv (ccsh VV) (csh A) W.
ccovv : ({w} ccofv (Vf w) (Af w) w) -> ccofvv (ccvv Vf) Af.

% freeze a regular variable within an expression
freeze : {N  : ccval -> ccexp}
         {N' : ccval -> ccexp}
         {F  : frozen N'} type.              %name freeze F f.
%mode freeze +D -D' -F'.

vfreeze : {N  : ccval -> ccval}
          {N' : ccval -> ccval}
          {F  : vfrozen N'} type.            %name vfreeze F f.
%mode vfreeze +D -D' -F'.

vvfreeze : {N  : ccval -> ccvval}
           {N' : ccval -> ccvval}
           {F  : vvfrozen N'} type.          %name vvfreeze F f.
%mode vvfreeze +D -D' -F'.

fz/closed : vfreeze ([v] V) ([v] V) vf/closed.
fz/halt : freeze ([v] cchalt) ([v] cchalt) f/halt.
fz/fst : freeze ([v] ccfst (V v) ([x] N x v))
                ([v] ccfst (V' v) ([x] N' x v))
                (f/fst FV FN)
      <- vfreeze V V' FV
      <- ({x} freeze (N x) (N' x) (FN x)).

fz/snd : freeze ([v] ccsnd (V v) ([x] N x v))
                ([v] ccsnd (V' v) ([x] N' x v))
                (f/snd FV FN)
      <- vfreeze V V' FV
      <- ({x} freeze (N x) (N' x) (FN x)).

fz/call : freeze ([v] cccall (V1 v) (V2 v))
                 ([v] cccall (V1' v) (V2' v))
                 (f/call FV1 FV2)
         <- vfreeze V1 V1' FV1
         <- vfreeze V2 V2' FV2.

fz/go : freeze ([v] ccgo W (V1 v) (V2 v))
               ([v] ccgo W (V1' v) (V2' v))
               (f/go FV1 FV2)
     <- vfreeze V1 V1' FV1
     <- vfreeze V2 V2' FV2.

fz/wapp : freeze ([v] ccwapp (V v) W ([x] N x v))
                 ([v] ccwapp (V' v) W ([x] N' x v))
                 (f/wapp FV FN)
       <- vfreeze V V' FV
       <- ({x} freeze (N x) (N' x) (FN x)).

fz/unpack : freeze ([v] ccunpack (V v) ([w][x] N w x v))
                   ([v] ccunpack (V' v) ([w][x] N' w x v))
                   (f/unpack FV FN)
         <- vfreeze V V' FV
         <- ({w}{x} freeze (N w x) (N' w x) (FN w x)).

fz/leta : freeze ([v] ccleta (V v) ([x] N x v))
                 ([v] ccleta (V' v) ([x] N' x v))
                 (f/leta FV FN)
       <- vfreeze V V' FV
       <- ({x} freeze (N x) (N' x) (FN x)).

fz/localhost : freeze ([v] cclocalhost ([x] N x v))
                      ([v] cclocalhost ([x] N' x v))
```

```
                    (f/localhost FN)
           <- ({x} freeze (N x) (N' x) (FN x)).

fz/lets : freeze ([v] cclets (V v) ([xx] N xx v))
                    ([v] cclets (V' v) ([xx] N' xx v))
                    (f/lets FV FN)
          <- vfreeze V V' FV
          <- ({xx} freeze (N xx) (N' xx) (FN xx)).

fz/put : freeze ([v] ccput (V v) ([xx] N xx v))
                ([v] ccput (V' v) ([xx] N' xx v))
                (f/put FV FN)
       <- vfreeze V V' FV
       <- ({xx} freeze (N xx) (N' xx) (FN xx)).

fz/valid : vfreeze ([v] ccvalid (VV v)) ([v] ccvalid (VV' v))
                  (vf/valid FVV)
       <- vvfreeze VV VV' FVV.

fz/pair : vfreeze ([v] ccpair (V1 v) (V2 v))
                  ([v] ccpair (V1' v) (V2' v))
                  (vf/pair FV1 FV2)
       <- vfreeze V1 V1' FV1
       <- vfreeze V2 V2' FV2.

fz/held : vfreeze ([v] ccheld (V v)) ([v] ccheld (V' v))
                  (vf/held FV)
       <- vfreeze V V' FV.

fz/const : vfreeze ([v] ccconst W) ([v] ccconst W) vf/const.
fz/1 : vfreeze ([v] cc1) ([v] cc1) vf/1.

fz/wlam : vfreeze ([x] ccwlam ([w] V w x)) ([x] ccwlam ([w] V' w x))
                  (vf/wlam FV)
       <- ({w} vfreeze (V w) (V' w) (FV w)).

fz/pack : vfreeze ([v] ccpack W (V v)) ([v] ccpack W (V' v))
                  (vf/pack FV)
       <- vfreeze V V' FV.

% this is the only interesting case of freeze.
fz/closure :
  vfreeze ([x:ccval] ccclosure ([a:ccval][e:ccval] BOD a e x) (ENV x))
            % prepend body code with projection of the variable
            % from the environment; add variable as fst projection
            % of environment itself. (It is "held" so that it can
            % be at any other world.)
            ([x] ccclosure ([a:ccval][e:ccval]
                              ccfst e [exh:ccval]
                              ccsnd e [envtail:ccval]
                              ccleta exh [ex:ccval]
                              BOD' a envtail ex) (ccpair (ccheld x) (ENV' x)))
            (vf/closure (vf/pair (vf/held vf/var) FENV))
          <- ({a:ccval}{e:ccval} freeze ([x] BOD a e x) ([x] BOD' a e x) _)
          <- vfreeze ENV ENV' FENV.

fz/var : vfreeze ([v] v) ([v] v) vf/var.

fz/ch : vfreeze ([v] ccsh (VV v)) ([v] ccsh (VV' v)) (vf/ch FVV)
     <- vvfreeze VV VV' FVV.

fz/vv : vvfreeze ([v] ccvv ([w] V w v)) ([v] ccvv ([w] V' w v)) (vvf/vv FVV)
     <- ({w} vfreeze (V w) (V' w) (FVV w)).

fz/vclosed : vvfreeze ([v] VV) ([v] VV) vvf/closed.
```

```
% mostly duplicated: same now for valid variables
% freeze a regular variable within an expression
freezevv : {N  : ccvval -> ccexp}
           {N' : ccvval -> ccexp}
           {F  : frozenvv N'}
           type.
%mode freezevv +D -D' -F'.

vfreezevv : {N  : ccvval -> ccval}
            {N' : ccvval -> ccval}
            {F  : vfrozenvv N'}
            type.
%mode vfreezevv +D -D' -F'.

vvfreezevv : {N  : ccvval -> ccvval}
             {N' : ccvval -> ccvval}
             {F  : vvfrozenvv N'}
             type.
%mode vvfreezevv +D -D' -F'.

fzvv/halt : freezevv ([v] cchalt) ([v] cchalt) fvv/halt.
fzvv/fst : freezevv ([v] ccfst (V v) ([x] N x v))
                ([v] ccfst (V' v) ([x] N' x v))
                (fvv/fst FV FN)
      <- vfreezevv V V' FV
      <- ({x} freezevv (N x) (N' x) (FN x)).
fzvv/snd : freezevv ([v] ccsnd (V v) ([x] N x v))
                ([v] ccsnd (V' v) ([x] N' x v))
                (fvv/snd FV FN)
      <- vfreezevv V V' FV
      <- ({x} freezevv (N x) (N' x) (FN x)).
fzvv/call : freezevv ([v] cccall (V1 v) (V2 v))
                 ([v] cccall (V1' v) (V2' v))
                 (fvv/call FV1 FV2)
         <- vfreezevv V1 V1' FV1
         <- vfreezevv V2 V2' FV2.
fzvv/go : freezevv ([v] ccgo W (V1 v) (V2 v))
               ([v] ccgo W (V1' v) (V2' v))
               (fvv/go FV1 FV2)
     <- vfreezevv V1 V1' FV1
     <- vfreezevv V2 V2' FV2.
fzvv/wapp : freezevv ([v] ccwapp (V v) W ([x] N x v))
                 ([v] ccwapp (V' v) W ([x] N' x v))
                 (fvv/wapp FV FN)
       <- vfreezevv V V' FV
       <- ({x} freezevv (N x) (N' x) (FN x)).
fzvv/unpack : freezevv ([v] ccunpack (V v) ([w][x] N w x v))
                   ([v] ccunpack (V' v) ([w][x] N' w x v))
                   (fvv/unpack FV FN)
         <- vfreezevv V V' FV
         <- ({w}{x} freezevv (N w x) (N' w x) (FN w x)).
fzvv/leta : freezevv ([v] ccleta (V v) ([x] N x v))
                 ([v] ccleta (V' v) ([x] N' x v))
                 (fvv/leta FV FN)
       <- vfreezevv V V' FV
       <- ({x} freezevv (N x) (N' x) (FN x)).
fzvv/localhost : freezevv ([v] cclocalhost ([x] N x v))
                      ([v] cclocalhost ([x] N' x v))
                      (fvv/localhost FN)
           <- ({x} freezevv (N x) (N' x) (FN x)).
fzvv/lets : freezevv ([v] cclets (V v) ([xx] N xx v))
                     ([v] cclets (V' v) ([xx] N' xx v))
                     (fvv/lets FV FN)
           <- vfreezevv V V' FV
           <- ({xx} freezevv (N xx) (N' xx) (FN xx)).
```

```
fzvv/put : freezevv ([v] ccput (V v) ([xx] N xx v))
                 ([v] ccput (V' v) ([xx] N' xx v))
                 (fvv/put FV FN)
       <- vfreezevv V V' FV
       <- ({xx} freezevv (N xx) (N' xx) (FN xx)).
fzvv/valid : vfreezevv ([v] ccvalid (VV v))
                  ([v] ccvalid (VV' v))
                  (vfvv/valid FVV)
       <- vvfreezevv VV VV' FVV.
fzvv/pair : vfreezevv ([v] ccpair (V1 v) (V2 v))
                 ([v] ccpair (V1' v) (V2' v))
                 (vfvv/pair FV1 FV2)
       <- vfreezevv V1 V1' FV1
       <- vfreezevv V2 V2' FV2.
fzvv/held : vfreezevv ([v] ccheld (V v))
                 ([v] ccheld (V' v))
                 (vfvv/held FV)
       <- vfreezevv V V' FV.
fzvv/const : vfreezevv ([v] ccconst W) ([v] ccconst W) vfvv/const.
fzvv/1 : vfreezevv ([v] cc1) ([v] cc1) vfvv/1.
fzvv/wlam : vfreezevv ([x] ccwlam ([w] V w x))
                 ([x] ccwlam ([w] V' w x))
                 (vfvv/wlam FV)
       <- ({w} vfreezevv (V w) (V' w) (FV w)).
fzvv/pack : vfreezevv ([v] ccpack W (V v))
                 ([v] ccpack W (V' v))
                 (vfvv/pack FV)
      <- vfreezevv V V' FV.
fzvv/closedvv : vvfreezevv ([v] V) ([v] V) vvfvv/closed.
fzvv/ch : vfreezevv ([v] ccsh (VV v)) ([v] ccsh (VV' v)) (vfvv/ch FVV)
       <- vvfreezevv VV VV' FVV.
fzvv/vv : vvfreezevv ([v] ccvv ([w] V w v)) ([v] ccvv ([w] V' w v)) (vvfvv/vv FVV)
     <- ({w} vfreezevv (V w) (V' w) (FVV w)).
fzvv/closed : vfreezevv ([vv] V) ([vv] V) vfvv/closed.
fzvv/var : vvfreezevv ([u] u) ([u] u) vvfvv/var.
%% end duplicated

% this case is different.
fzvv/closure : vfreezevv ([u:ccvval] ccclosure ([a:ccval][e:ccval] BOD a e u) (ENV u))
                    % prepend body code with projection of the variable from the
                    % environment; add variable as fst projection of environment
                    % itself. The variable is injected into the sh modality.
                    ([u] ccclosure ([a:ccval][e:ccval]
                                    ccfst e [ehu:ccval]
                                    ccsnd e [ee:ccval]
                                    cclets ehu [u:ccvval]
                                    BOD' a ee u) (ccpair (ccsh u) (ENV' u)))
                    (vfvv/closure (vfvv/pair (vfvv/ch vvfvv/var) FENV))
         <- ({a:ccval}{e:ccval} freezevv ([u] BOD a e u) ([u] BOD' a e u) _)
         <- vfreezevv ENV ENV' FENV.

%worlds (blockw | blockccv | blockccvv) (freeze _ _ _) (vfreeze _ _ _) (vvfreeze _ _ _).
%total (D E F) (freeze D _ _) (vfreeze E _ _) (vvfreeze F _ _).

%worlds (blockw | blockccv | blockccvv) (freezevv _ _ _) (vfreezevv _ _ _) (vvfreezevv _ _ _).
%total (D E F) (freezevv D _ _) (vfreezevv E _ _) (vvfreezevv F _ _).

% some equalities necessary now...
ccval-eq : ccval -> ccval -> type.
ccval-eq/ : ccval-eq V V.

ccexp-eq : ccexp -> ccexp -> type.
ccexp-eq/ : ccexp-eq V V.

ccvval-eq : ccvval -> ccvval -> type.
ccvval-eq/ : ccvval-eq V V.
```

```
frozen-resp : ({x} ccexp-eq (V x) (V' x)) -> frozen V -> frozen V' -> type.
%mode frozen-resp +E +F -F'.
- : frozen-resp ([_] ccexp-eq/) F F.

vfrozen-resp : ({x} ccval-eq (V x) (V' x)) -> vfrozen V -> vfrozen V' -> type.
%mode vfrozen-resp +E +F -F'.
- : vfrozen-resp ([_] ccval-eq/) F F.

vvfrozen-resp : ({x} ccvval-eq (V x) (V' x)) -> vvfrozen V -> vvfrozen V' -> type.
%mode vvfrozen-resp +E +F -F'.
- : vvfrozen-resp ([_] ccvval-eq/) F F.

frozenvv-resp : ({x} ccexp-eq (V x) (V' x)) -> frozenvv V -> frozenvv V' -> type.
%mode frozenvv-resp +E +F -F'.
- : frozenvv-resp ([_] ccexp-eq/) F F.

vfrozenvv-resp : ({x} ccval-eq (V x) (V' x)) -> vfrozenvv V -> vfrozenvv V' -> type.
%mode vfrozenvv-resp +E +F -F'.
- : vfrozenvv-resp ([_] ccval-eq/) F F.

vvfrozenvv-resp : ({x} ccvval-eq (V x) (V' x)) -> vvfrozenvv V -> vvfrozenvv V' -> type.
%mode vvfrozenvv-resp +E +F -F'.
- : vvfrozenvv-resp ([_] ccvval-eq/) F F.

%worlds (blockw | blockccv | blockccvv) (frozenvv-resp _ _ _) (vfrozenvv-resp _ _ _)
  (vvfrozenvv-resp _ _ _) (frozen-resp _ _ _) (vfrozen-resp _ _ _) (vvfrozen-resp _ _ _).
%total D (frozenvv-resp D _ _).
%total D (vfrozenvv-resp D _ _).
%total D (vvfrozenvv-resp D _ _).
%total D (frozen-resp D _ _).
%total D (vfrozen-resp D _ _).
%total D (vvfrozen-resp D _ _).

% freezing needs to preserve closedness
permaclosed : {F : {v:ccval} freeze ([x] N x) ([x] N' v x) (Z v)}
              {E : {x} {y} ccexp-eq (N'' y) (N' x y)} type.
%mode permaclosed +F -E.

permavclosed : {F : {v:ccval} vfreeze ([x] V x) ([x] V' v x) (Z v)}
               {E : {x} {y} ccval-eq (V'' y) (V' x y)} type.
%mode permavclosed +F -E.

permavvclosed : {F : {v:ccval} vvfreeze ([x] VV x) ([x] VV' v x) (Z v)}
                {E : {x} {y} ccvval-eq (VV'' y) (VV' x y)} type.
%mode permavvclosed +F -E.

% for closedness wrt vvals
vvpermaclosed : {F : {v:ccvval} freeze ([x:ccval] V x) ([x:ccval] V' v x) (Z v)}
                {E : {x} {y} ccexp-eq (V'' y) (V' x y)} type.
%mode vvpermaclosed +F -E.

vvpermavclosed : {F : {v:ccvval} vfreeze ([x:ccval] V x) ([x:ccval] V' v x) (Z v)}
                 {E : {x} {y} ccval-eq (V'' y) (V' x y)} type.
%mode vvpermavclosed +F -E.

vvpermavvclosed : {F : {v:ccvval} vvfreeze ([x:ccval] V x) ([x:ccval] V' v x) (Z v)}
                  {E : {x} {y} ccvval-eq (V'' y) (V' x y)} type.
%mode vvpermavvclosed +F -E.

cclocalhost-resp : ({x} ccexp-eq (N x) (N' x)) ->
                   ccexp-eq (cclocalhost N) (cclocalhost N') -> type.
%mode cclocalhost-resp +D -E.
- : cclocalhost-resp ([_] ccexp-eq/) ccexp-eq/.

ccput-resp : ccval-eq V V' -> ({x} ccexp-eq (N x) (N' x)) ->
```

```
             ccexp-eq (ccput V N) (ccput V' N') -> type.
%mode ccput-resp +D1 +D -E.
- : ccput-resp ccval-eq/ ([_] ccexp-eq/) ccexp-eq/.

ccunpack-resp : ccval-eq V V' -> ({w}{x} ccexp-eq (N w x) (N' w x)) ->
                ccexp-eq (ccunpack V N) (ccunpack V' N') -> type.
%mode ccunpack-resp +D1 +D -E.
- : ccunpack-resp ccval-eq/ ([_][_] ccexp-eq/) ccexp-eq/.

cclets-resp : ccval-eq V V' -> ({x} ccexp-eq (N x) (N' x)) ->
              ccexp-eq (cclets V N) (cclets V' N') -> type.
%mode cclets-resp +D1 +D -E.
- : cclets-resp ccval-eq/ ([_] ccexp-eq/) ccexp-eq/.

ccbind-resp : ({x} ccexp-eq (N x) (N' x)) ->
              {F : (ccval -> ccexp) -> ccexp}
              ccexp-eq (F N) (F N') -> type.
%mode ccbind-resp +D +F -E.
- : ccbind-resp ([_] ccexp-eq/) _ ccexp-eq/.

ccubind-resp : ({x} ccexp-eq (N x) (N' x)) ->
               {F : (ccvval -> ccexp) -> ccexp}
               ccexp-eq (F N) (F N') -> type.
%mode ccubind-resp +D +F -E.
- : ccubind-resp ([_] ccexp-eq/) _ ccexp-eq/.

ccvalbind-resp : ccval-eq V V' -> ({x} ccexp-eq (N x) (N' x)) ->
                 {F : ccval -> (ccval -> ccexp) -> ccexp}
                 ccexp-eq (F V N) (F V' N') -> type.
%mode ccvalbind-resp +D1 +D +F -E.
- : ccvalbind-resp ccval-eq/ ([_] ccexp-eq/) _ ccexp-eq/.

ccvalubind-resp : ccval-eq V V' -> ({x} ccexp-eq (N x) (N' x)) ->
                  {F : ccval -> (ccvval -> ccexp) -> ccexp}
                  ccexp-eq (F V N) (F V' N') -> type.
%mode ccvalubind-resp +D1 +D +F -E.
- : ccvalubind-resp ccval-eq/ ([_] ccexp-eq/) _ ccexp-eq/.

cccall-resp : ccval-eq V1 V1' -> ccval-eq V2 V2' ->
              ccexp-eq (cccall V1 V2) (cccall V1' V2') -> type.
%mode cccall-resp +D1 +D2 -D3.
- : cccall-resp ccval-eq/ ccval-eq/ ccexp-eq/.

cc2val-resp : ccval-eq V1 V1' -> ccval-eq V2 V2' ->
              {F : ccval -> ccval -> ccexp}
              ccexp-eq (F V1 V2) (F V1' V2') -> type.
%mode cc2val-resp +D1 +D2 +F -D3.
- : cc2val-resp ccval-eq/ ccval-eq/ _ ccexp-eq/.

ccwlam-resp : ({w} ccval-eq (V w) (V' w)) ->
              ccval-eq (ccwlam V) (ccwlam V') ->
              type.
%mode ccwlam-resp +V -E.
- : ccwlam-resp ([_] ccval-eq/) ccval-eq/.

ccvv-resp : ({w} ccval-eq (V w) (V' w)) ->
              ccvval-eq (ccvv V) (ccvv V') ->
              type.
%mode ccvv-resp +V -E.
- : ccvv-resp ([_] ccval-eq/) ccvval-eq/.

ccclosure-resp : ({a}{e} ccexp-eq (E a e) (E' a e)) ->
                 ccval-eq V V' ->
                 ccval-eq (ccclosure E V) (ccclosure E' V') ->
                 type.
%mode ccclosure-resp +E +V -D.
```

```
- : ccclosure-resp ([_][_] ccexp-eq/) ccval-eq/ ccval-eq/.

ccsh-resp : ccvval-eq VV VV' ->
            ccval-eq (ccsh VV) (ccsh VV') -> type.
%mode ccsh-resp +D1 -D.
- : ccsh-resp ccvval-eq/ ccval-eq/.

ccvalid-resp : ccvval-eq VV VV' ->
            ccval-eq (ccvalid VV) (ccvalid VV') -> type.
%mode ccvalid-resp +D1 -D.
- : ccvalid-resp ccvval-eq/ ccval-eq/.

cc2valv-resp : ccval-eq V1 V1' -> ccval-eq V2 V2' ->
               {F : ccval -> ccval -> ccval}
               ccval-eq (F V1 V2) (F V1' V2') -> type.
%mode cc2valv-resp +D1 +D2 +F -D3.
- : cc2valv-resp ccval-eq/ ccval-eq/ _ ccval-eq/.

ccuv-resp : ccvval-eq V1 V1' -> ccval-eq V2 V2' ->
            {F : ccvval -> ccval -> ccval}
            ccval-eq (F V1 V2) (F V1' V2') -> type.
%mode ccuv-resp +D1 +D2 +F -D3.
- : ccuv-resp ccvval-eq/ ccval-eq/ _ ccval-eq/.

ccf-resp : ccexp-eq N N' -> {F : ccexp -> ccexp}
           ccexp-eq (F N) (F N') -> type.
%mode ccf-resp +D +F -D'.
- : ccf-resp ccexp-eq/ _ ccexp-eq/.

% Since equalities are so trivial to prove, we can prove very strong ones:
ccf4-resp : ({a}{b}{c}{d} ccexp-eq (N a b c d) (N' a b c d)) ->
            {F : (ccval -> ccval -> ccval -> ccval -> ccexp) -> ccval}
            ccval-eq (F N) (F N') -> type.
%mode ccf4-resp +D +F -D'.
- : ccf4-resp ([_] [_] [_] [_] ccexp-eq/) _ ccval-eq/.

ccf1u3-resp : ({a}{b}{c}{d} ccexp-eq (N a b c d) (N' a b c d)) ->
              {F : (ccval -> ccvval -> ccval -> ccval -> ccexp) -> ccval}
              ccval-eq (F N) (F N') -> type.
%mode ccf1u3-resp +D +F -D'.
- : ccf1u3-resp ([_] [_] [_] [_] ccexp-eq/) _ ccval-eq/.

ccfu3-resp : ({a}{b}{c}{d} ccexp-eq (N a b c d) (N' a b c d)) ->
             {F : (ccvval -> ccval -> ccval -> ccval -> ccexp) -> ccval}
             ccval-eq (F N) (F N') -> type.
%mode ccfu3-resp +D +F -D'.
- : ccfu3-resp ([_] [_] [_] [_] ccexp-eq/) _ ccval-eq/.

ccfuu2-resp : ({a}{b}{c}{d} ccexp-eq (N a b c d) (N' a b c d)) ->
              {F : (ccvval -> ccvval -> ccval -> ccval -> ccexp) -> ccval}
              ccval-eq (F N) (F N') -> type.
%mode ccfuu2-resp +D +F -D'.
- : ccfuu2-resp ([_] [_] [_] [_] ccexp-eq/) _ ccval-eq/.

%worlds (blockw | blockccv | blockccvv) (cclocalhost-resp _ _)
  (ccclosure-resp _ _ _) (ccput-resp _ _ _) (ccunpack-resp _ _ _)
  (cclets-resp _ _ _) (ccbind-resp _ _ _) (ccubind-resp _ _ _) (ccvalbind-resp _ _ _ _)
  (ccvalubind-resp _ _ _ _) (cc2val-resp _ _ _ _) (cc2valv-resp _ _ _ _) (ccuv-resp _ _ _ _)
  (ccsh-resp _ _) (ccwlam-resp _ _) (ccvv-resp _ _) (ccf-resp _ _ _) (ccf4-resp _ _ _)
  (ccfu3-resp _ _ _) (ccfuu2-resp _ _ _) (ccvalid-resp _ _) (cccall-resp _ _ _).
%total D (cclocalhost-resp D _).
%total D (cccall-resp D _ _).
%total D (ccput-resp D _ _).
%total D (ccunpack-resp D _ _).
%total D (cclets-resp D _ _).
%total D (ccvalbind-resp D _ _ _).
```

```
%total D (ccbind-resp D _ _).
%total D (ccubind-resp D _ _).
%total D (cc2val-resp D _ _ _).
%total D (cc2valv-resp D _ _ _).
%total D (ccuv-resp D _ _ _).
%total D (ccsh-resp D _).
%total D (ccf-resp D _ _).
%total D (ccf4-resp D _ _).
%total D (ccfu3-resp D _ _).
%total D (ccfuu2-resp D _ _).
%total D (ccclosure-resp D _ _).
%total D (ccwlam-resp D _).
%total D (ccvalid-resp D _).
%total D (ccvalubind-resp D _ _ _).
%total D (ccvv-resp D _).

- : permaclosed ([_] fz/halt) ([_][_] ccexp-eq/).
- : permaclosed ([x] fz/localhost (F x)) EQ'
 <- ({y:ccval} permaclosed ([x] F x y) ([v] [x] EQ v x y))
 <- ({a} {x} cclocalhost-resp ([y] EQ a x y) (EQ' a x)).
- : permaclosed ([x] fz/call (FA x) (FV x)) EQ'
 <- permavclosed FA EQA
 <- permavclosed FV EQV
 <- ({a} {x} cc2val-resp (EQV a x) (EQA a x) cccall (EQ' a x)).
- : permaclosed ([x] fz/put (FN x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({y} permaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccput-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : permaclosed ([x] fz/lets (FN x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({y} permaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} cclets-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : permaclosed ([x] fz/leta (FN x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({y} permaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccleta (EQ' a x)).
- : permaclosed ([x] fz/unpack ([w] FN w x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({w:world} {y} permaclosed ([x] FN w x y) ([v][x] EQN w v x y))
 <- ({a} {x} ccunpack-resp (EQV a x) ([w][y] EQN w a x y) (EQ' a x)).
- : permaclosed ([x] fz/wapp (FN x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({y} permaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ([a][x] ccwapp a W x) (EQ' a x)).
- : permaclosed ([x] fz/fst (FN x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({y} permaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccfst (EQ' a x)).
- : permaclosed ([x] fz/snd (FN x) (FV x)) EQ'
 <- permavclosed FV EQV
 <- ({y} permaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccsnd (EQ' a x)).
- : permaclosed ([x] fz/go (FA x) (FV x)) EQ'
 <- permavclosed FA EQA
 <- permavclosed FV EQV
 <- ({a} {x} cc2val-resp (EQA a x) (EQV a x) ([a][b] ccgo W b a) (EQ' a x)).

- : permavclosed ([_] fz/var) ([_][_] ccval-eq/).
- : permavclosed ([_] fz/closed) ([_][_] ccval-eq/).
- : permavclosed ([x] fz/pair (F2 x) (F1 x)) EQ'
 <- permavclosed F1 EQ1
 <- permavclosed F2 EQ2
 <- ({a} {x} cc2valv-resp (EQ1 a x) (EQ2 a x) ccpair (EQ' a x)).
- : permavclosed ([x] fz/held (F x)) EQ'
 <- permavclosed F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccheld 1) (EQ' a x)).
```

```
- : permavclosed ([_] fz/1) ([_][_] ccval-eq/).
- : permavclosed ([_] fz/const) ([_][_] ccval-eq/).
- : permavclosed ([x] fz/ch (F x)) EQ'
 <- permavvclosed F EQ
 <- ({a} {x} ccsh-resp (EQ a x) (EQ' a x)).
- : permavclosed ([x] fz/pack (F x)) EQ'
 <- permavclosed F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccpack W 1) (EQ' a x)).
- : permavclosed ([x] fz/wlam ([w] F x w)) EQ'
 <- ({w} permavclosed ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccwlam-resp ([w] EQ a x w) (EQ' a x)).
- : permavclosed ([x] fz/valid (FF x)) EQ'
 <- permavvclosed FF EQ
 <- ({a} {x} ccvalid-resp (EQ a x) (EQ' a x)).
- : permavvclosed ([x] fz/vv ([w] F x w)) EQ'
 <- ({w} permavclosed ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccvv-resp ([w] EQ a x w) (EQ' a x)).
- : permavvclosed ([_] fz/vclosed) ([_][_] ccvval-eq/).

% mainly tricky from keeping track of all of the free variables..
- : permavclosed ([x] fz/closure (FE x) (FB x)) EQ'
 <- permavclosed FE ([a][x] (EQE a x) : ccval-eq (ENV'' x) (ENV' a x))
 <- ({toss} {x} cc2valv-resp (ccval-eq/ : ccval-eq x _) (EQE toss x) ([y][e] ccpair (ccheld y) e)
       (EQE2 toss x))
 <- ({arg} {envtail}
       permaclosed ([x] FB x arg envtail)
          ([toss] [x] EQB toss x arg envtail :
                ccexp-eq (BOD'' x arg envtail) (BOD' toss x arg envtail)))
 <- ({arg} {envtail} {toss} {exh}
       % unused eq
       ccvalbind-resp (ccval-eq/ : ccval-eq cc1 cc1)
                      ([ex] EQB toss ex arg envtail :
                        ccexp-eq (BOD'' ex arg envtail) (BOD' toss ex arg envtail))
                      ([_][n] ccleta exh n) (EQB2 arg envtail toss exh))
 <- ({arg} {toss} {exh} {e}
       % unused eq
       ccvalbind-resp (ccval-eq/ : ccval-eq cc1 cc1)
       ([envtail] EQB2 arg envtail toss exh) ([_][n] ccsnd e n) (EQB3 arg toss exh e))
 <- ({arg} {toss} {e}
       ccvalbind-resp (ccval-eq/ : ccval-eq cc1 cc1)
       ([exh] EQB3 arg toss exh e) ([_][n] ccfst e n) (EQB4 arg toss e))
 <- ({toss} {x}
       ccclosure-resp ([arg][e] EQB4 arg toss e) (EQE2 toss x) (EQ' toss x)).

%worlds (blockw | blockccv | blockccvv) (permavclosed _ _) (permaclosed _ _) (permavvclosed _ _).
%total (D E F) (permaclosed D _) (permavclosed E _) (permavvclosed F _).

% this is duplicated from the above with s/perma/vvperma
- : vvpermaclosed ([_] fz/halt) ([_][_] ccexp-eq/).
- : vvpermaclosed ([x] fz/localhost (F x)) EQ'
 <- ({y:ccval} vvpermaclosed ([x] F x y) ([v] [x] EQ v x y))
 <- ({a} {x} cclocalhost-resp ([y] EQ a x y) (EQ' a x)).
- : vvpermaclosed ([x] fz/call (FA x) (FV x)) EQ'
 <- vvpermavclosed FA EQA
 <- vvpermavclosed FV EQV
 <- ({a} {x} cc2val-resp (EQV a x) (EQA a x) cccall (EQ' a x)).
- : vvpermaclosed ([x] fz/put (FN x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
 <- ({y} vvpermaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccput-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : vvpermaclosed ([x] fz/lets (FN x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
 <- ({y} vvpermaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} cclets-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : vvpermaclosed ([x] fz/leta (FN x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
```

```
 <- ({y} vvpermaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccleta (EQ' a x)).
- : vvpermaclosed ([x] fz/unpack ([w] FN w x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
 <- ({w:world} {y} vvpermaclosed ([x] FN w x y) ([v][x] EQN w v x y))
 <- ({a} {x} ccunpack-resp (EQV a x) ([w][y] EQN w a x y) (EQ' a x)).
- : vvpermaclosed ([x] fz/wapp (FN x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
 <- ({y} vvpermaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ([a][x] ccwapp a W x) (EQ' a x)).
- : vvpermaclosed ([x] fz/fst (FN x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
 <- ({y} vvpermaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccfst (EQ' a x)).
- : vvpermaclosed ([x] fz/snd (FN x) (FV x)) EQ'
 <- vvpermavclosed FV EQV
 <- ({y} vvpermaclosed ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccsnd (EQ' a x)).
- : vvpermaclosed ([x] fz/go (FA x) (FV x)) EQ'
 <- vvpermavclosed FA EQA
 <- vvpermavclosed FV EQV
 <- ({a} {x} cc2val-resp (EQA a x) (EQV a x) ([a][b] ccgo W b a) (EQ' a x)).
- : vvpermavclosed ([_] fz/var) ([_][_] ccval-eq/).
- : vvpermavclosed ([_] fz/closed) ([_][_] ccval-eq/).
- : vvpermavclosed ([x] fz/pair (F2 x) (F1 x)) EQ'
 <- vvpermavclosed F1 EQ1
 <- vvpermavclosed F2 EQ2
 <- ({a} {x} cc2valv-resp (EQ1 a x) (EQ2 a x) ccpair (EQ' a x)).
- : vvpermavclosed ([x] fz/held (F x)) EQ'
 <- vvpermavclosed F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccheld 1) (EQ' a x)).
- : vvpermavclosed ([_] fz/1) ([_][_] ccval-eq/).
- : vvpermavclosed ([_] fz/const) ([_][_] ccval-eq/).
- : vvpermavclosed ([x] fz/ch (F x)) EQ'
 <- vvpermavvclosed F EQ
 <- ({a} {x} ccsh-resp (EQ a x) (EQ' a x)).
- : vvpermavclosed ([x] fz/pack (F x)) EQ'
 <- vvpermavclosed F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccpack W 1) (EQ' a x)).
- : vvpermavclosed ([x] fz/wlam ([w] F x w)) EQ'
 <- ({w} vvpermavclosed ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccwlam-resp ([w] EQ a x w) (EQ' a x)).
- : vvpermavclosed ([x] fz/valid (FF x)) EQ'
 <- vvpermavvclosed FF EQ
 <- ({a} {x} ccvalid-resp (EQ a x) (EQ' a x)).
- : vvpermavvclosed ([x] fz/vv ([w] F x w)) EQ'
 <- ({w} vvpermavclosed ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccvv-resp ([w] EQ a x w) (EQ' a x)).
- : vvpermavvclosed ([_] fz/vclosed) ([_][_] ccvval-eq/).
- : vvpermavclosed ([x] fz/closure (FE x) (FB x)) EQ'
 <- vvpermavclosed FE ([a][x] (EQE a x) : ccval-eq (ENV'' x) (ENV' a x))
 <- ({toss} {x} cc2valv-resp (ccval-eq/ : ccval-eq x _) (EQE toss x) ([y][e] ccpair (ccheld y) e)
                            (EQE2 toss x))
 <- ({arg} {envtail}
       vvpermaclosed ([x] FB x arg envtail)
         ([toss] [x] EQB toss x arg envtail
           : ccexp-eq (BOD'' x arg envtail) (BOD' toss x arg envtail)))
 <- ({arg} {envtail} {toss} {exh}
       ccvalbind-resp (ccval-eq/ : ccval-eq cc1 cc1)
         ([ex] EQB toss ex arg envtail :
            ccexp-eq (BOD'' ex arg envtail) (BOD' toss ex arg envtail))
         ([_][n] ccleta exh n) (EQB2 arg envtail toss exh))
 <- ({arg} {toss} {exh} {e}
       ccvalbind-resp (ccval-eq/ : ccval-eq cc1 cc1)
       ([envtail] EQB2 arg envtail toss exh) ([_][n] ccsnd e n) (EQB3 arg toss exh e))
 <- ({arg} {toss} {e}
```

```
       ccvalbind-resp (ccval-eq/ : ccval-eq cc1 cc1) ([exh] EQB3 arg toss exh e)
       ([_][n] ccfst e n) (EQB4 arg toss e))
 <- ({toss} {x}
       ccclosure-resp ([arg][e] EQB4 arg toss e) (EQE2 toss x) (EQ' toss x)).

%worlds (blockw | blockccv | blockccvv)
        (vvpermavclosed _ _) (vvpermaclosed _ _) (vvpermavvclosed _ _).
%total (D E F) (vvpermaclosed D _) (vvpermavclosed E _) (vvpermavvclosed F _).
%% end duplication

% freezevving needs to preserve closedness
permaclosedvv : {F : {v:ccval} freezevv ([x] N x) ([x] N' v x) (Z v)}
                {E : {x} {y} ccexp-eq (N'' y) (N' x y)} type.
%mode permaclosedvv +F -E.

permavclosedvv : {F : {v:ccval} vfreezevv ([x] V x) ([x] V' v x) (Z v)}
                 {E : {x} {y} ccval-eq (V'' y) (V' x y)} type.
%mode permavclosedvv +F -E.

permavvclosedvv : {F : {v:ccval} vvfreezevv ([x] VV x) ([x] VV' v x) (Z v)}
                  {E : {x} {y} ccvval-eq (VV'' y) (VV' x y)} type.
%mode permavvclosedvv +F -E.


% for closedvvness wrt vvals
vvpermaclosedvv : {F : {v:ccvval} freezevv ([x:ccvval] V x) ([x:ccvval] V' v x) (Z v)}
                  {E : {x} {y} ccexp-eq (V'' y) (V' x y)} type.
%mode vvpermaclosedvv +F -E.

vvpermavclosedvv : {F : {v:ccvval} vfreezevv ([x:ccvval] V x) ([x:ccvval] V' v x) (Z v)}
                   {E : {x} {y} ccval-eq (V'' y) (V' x y)} type.
%mode vvpermavclosedvv +F -E.

vvpermavvclosedvv : {F : {v:ccvval} vvfreezevv ([x:ccvval] V x) ([x:ccvval] V' v x) (Z v)}
                    {E : {x} {y} ccvval-eq (V'' y) (V' x y)} type.
%mode vvpermavvclosedvv +F -E.

%% duplicated from above
- : permaclosedvv ([_] fzvv/halt) ([_][_] ccexp-eq/).
- : permaclosedvv ([x] fzvv/localhost (F x)) EQ'
 <- ({y:ccval} permaclosedvv ([x] F x y) ([v] [x] EQ v x y))
 <- ({a} {x} cclocalhost-resp ([y] EQ a x y) (EQ' a x)).
- : permaclosedvv ([x] fzvv/call (FA x) (FV x)) EQ'
 <- permavclosedvv FA EQA
 <- permavclosedvv FV EQV
 <- ({a} {x} cc2val-resp (EQV a x) (EQA a x) cccall (EQ' a x)).
- : permaclosedvv ([x] fzvv/put (FN x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({y} permaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccput-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : permaclosedvv ([x] fzvv/lets (FN x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({y} permaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} cclets-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : permaclosedvv ([x] fzvv/leta (FN x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({y} permaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccleta (EQ' a x)).
- : permaclosedvv ([x] fzvv/unpack ([w] FN w x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({w:world} {y} permaclosedvv ([x] FN w x y) ([v][x] EQN w v x y))
 <- ({a} {x} ccunpack-resp (EQV a x) ([w][y] EQN w a x y) (EQ' a x)).
- : permaclosedvv ([x] fzvv/wapp (FN x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({y} permaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ([a][x] ccwapp a W x) (EQ' a x)).
```

```
- : permaclosedvv ([x] fzvv/fst (FN x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({y} permaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccfst (EQ' a x)).
- : permaclosedvv ([x] fzvv/snd (FN x) (FV x)) EQ'
 <- permavclosedvv FV EQV
 <- ({y} permaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccsnd (EQ' a x)).
- : permaclosedvv ([x] fzvv/go (FA x) (FV x)) EQ'
 <- permavclosedvv FA EQA
 <- permavclosedvv FV EQV
 <- ({a} {x} cc2val-resp (EQA a x) (EQV a x) ([a][b] ccgo W b a) (EQ' a x)).
- : permavclosedvv ([_] fzvv/closed) ([_][_] ccval-eq/).
- : permavclosedvv ([x] fzvv/pair (F2 x) (F1 x)) EQ'
 <- permavclosedvv F1 EQ1
 <- permavclosedvv F2 EQ2
 <- ({a} {x} cc2valv-resp (EQ1 a x) (EQ2 a x) ccpair (EQ' a x)).
- : permavclosedvv ([x] fzvv/held (F x)) EQ'
 <- permavclosedvv F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccheld 1) (EQ' a x)).
- : permavclosedvv ([_] fzvv/1) ([_][_] ccval-eq/).
- : permavclosedvv ([_] fzvv/const) ([_][_] ccval-eq/).
- : permavclosedvv ([x] fzvv/ch (F x)) EQ'
 <- permavvclosedvv F EQ
 <- ({a} {x} ccsh-resp (EQ a x) (EQ' a x)).
- : permavclosedvv ([x] fzvv/pack (F x)) EQ'
 <- permavclosedvv F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccpack W 1) (EQ' a x)).
- : permavclosedvv ([x] fzvv/wlam ([w] F x w)) EQ'
 <- ({w} permavclosedvv ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccwlam-resp ([w] EQ a x w) (EQ' a x)).
- : permavclosedvv ([x] fzvv/valid (FF x)) EQ'
 <- permavvclosedvv FF EQ
 <- ({a} {x} ccvalid-resp (EQ a x) (EQ' a x)).
- : permavvclosedvv ([x] fzvv/vv ([w] F x w)) EQ'
 <- ({w} permavclosedvv ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccvv-resp ([w] EQ a x w) (EQ' a x)).
%% end duplication; fzvv/closure is different here

- : permavclosedvv ([x] fzvv/closure (FE x) (FB x)) EQ'
 <- permavclosedvv FE ([a][x] (EQE a x) : ccval-eq (ENV'' x) (ENV' a x))
 <- ({toss} {x} ccuv-resp (ccvval-eq/ : ccvval-eq x _) (EQE toss x) ([y][e] ccpair (ccsh y) e)
                          (EQE2 toss x))
 <- ({arg} {envtail}
       permaclosedvv ([x] FB x arg envtail) ([toss] [x] EQB toss x arg envtail :
                        ccexp-eq (BOD'' x arg envtail) (BOD' toss x arg envtail)))
 <- ({arg} {envtail} {toss} {exh}
       ccubind-resp ([ex] EQB toss ex arg envtail : ccexp-eq (BOD'' ex arg envtail)
                     (BOD' toss ex arg envtail)) ([n] cclets exh n) (EQB2 arg envtail toss exh))
 <- ({arg} {toss} {exh} {e}
       ccbind-resp ([envtail] EQB2 arg envtail toss exh) ([n] ccsnd e n) (EQB3 arg toss exh e))
 <- ({arg} {toss} {e}
       ccbind-resp ([exh] EQB3 arg toss exh e) ([n] ccfst e n) (EQB4 arg toss e))
 <- ({toss} {x}
       ccclosure-resp ([arg][e] EQB4 arg toss e) (EQE2 toss x) (EQ' toss x)).

- : permavvclosedvv ([v] fzvv/closedvv) ([_][_] ccvval-eq/).
- : permavvclosedvv ([v] fzvv/var) ([_][_] ccvval-eq/).

%worlds (blockw | blockccv | blockccvv)
               (permavclosedvv _ _) (permaclosedvv _ _) (permavvclosedvv _ _).
%total (D E F) (permaclosedvv D _) (permavclosedvv E _) (permavvclosedvv F _).

- : vvpermaclosedvv ([_] fzvv/halt) ([_][_] ccexp-eq/).
- : vvpermaclosedvv ([x] fzvv/localhost (F x)) EQ'
 <- ({y:ccval} vvpermaclosedvv ([x] F x y) ([v] [x] EQ v x y))
```

```
 <- ({a} {x} cclocalhost-resp ([y] EQ a x y) (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/call (FA x) (FV x)) EQ'
 <- vvpermavclosedvv FA EQA
 <- vvpermavclosedvv FV EQV
 <- ({a} {x} cc2val-resp (EQV a x) (EQA a x) cccall (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/put (FN x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({y} vvpermaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccput-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/lets (FN x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({y} vvpermaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} cclets-resp (EQV a x) ([y] EQN a x y) (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/leta (FN x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({y} vvpermaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccleta (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/unpack ([w] FN w x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({w:world} {y} vvpermaclosedvv ([x] FN w x y) ([v][x] EQN w v x y))
 <- ({a} {x} ccunpack-resp (EQV a x) ([w][y] EQN w a x y) (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/wapp (FN x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({y} vvpermaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ([a][x] ccwapp a W x) (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/fst (FN x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({y} vvpermaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccfst (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/snd (FN x) (FV x)) EQ'
 <- vvpermavclosedvv FV EQV
 <- ({y} vvpermaclosedvv ([x] FN x y) ([v][x] EQN v x y))
 <- ({a} {x} ccvalbind-resp (EQV a x) ([y] EQN a x y) ccsnd (EQ' a x)).
- : vvpermaclosedvv ([x] fzvv/go (FA x) (FV x)) EQ'
 <- vvpermavclosedvv FA EQA
 <- vvpermavclosedvv FV EQV
 <- ({a} {x} cc2val-resp (EQA a x) (EQV a x) ([a][b] ccgo W b a) (EQ' a x)).
- : vvpermavclosedvv ([x] fzvv/pair (F2 x) (F1 x)) EQ'
 <- vvpermavclosedvv F1 EQ1
 <- vvpermavclosedvv F2 EQ2
 <- ({a} {x} cc2valv-resp (EQ1 a x) (EQ2 a x) ccpair (EQ' a x)).
- : vvpermavclosedvv ([x] fzvv/held (F x)) EQ'
 <- vvpermavclosedvv F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccheld 1) (EQ' a x)).
- : vvpermavclosedvv ([_] fzvv/1) ([_][_] ccval-eq/).
- : vvpermavclosedvv ([_] fzvv/const) ([_][_] ccval-eq/).
- : vvpermavclosedvv ([x] fzvv/ch (F x)) EQ'
 <- vvpermavvclosedvv F EQ
 <- ({a} {x} ccsh-resp (EQ a x) (EQ' a x)).
- : vvpermavclosedvv ([x] fzvv/pack (F x)) EQ'
 <- vvpermavclosedvv F EQ
 <- ({a} {x} cc2valv-resp (EQ a x) (EQ a x) ([1][2] ccpack W 1) (EQ' a x)).
- : vvpermavclosedvv ([x] fzvv/wlam ([w] F x w)) EQ'
 <- ({w} vvpermavclosedvv ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccwlam-resp ([w] EQ a x w) (EQ' a x)).
- : vvpermavclosedvv ([x] fzvv/valid (FF x)) EQ'
 <- vvpermavvclosedvv FF EQ
 <- ({a} {x} ccvalid-resp (EQ a x) (EQ' a x)).
- : vvpermavvclosedvv ([x] fzvv/vv ([w] F x w)) EQ'
 <- ({w} vvpermavclosedvv ([x] F x w) ([a][x] EQ a x w))
 <- ({a} {x} ccvv-resp ([w] EQ a x w) (EQ' a x)).
- : vvpermavclosedvv ([x] fzvv/closure (FE x) (FB x)) EQ'
 <- vvpermavclosedvv FE ([a][x] (EQE a x) : ccval-eq (ENV'' x) (ENV' a x))
 <- ({toss} {x} ccuv-resp (ccvval-eq/ : ccvval-eq x _) (EQE toss x) ([y][e] ccpair (ccsh y) e)
                          (EQE2 toss x))
 <- ({arg} {envtail}
```

```
       vvpermaclosedvv ([x] FB x arg envtail)
                       ([toss] [x] EQB toss x arg envtail :
                          ccexp-eq (BOD'' x arg envtail) (BOD' toss x arg envtail)))
 <- ({arg} {envtail} {toss} {exh}
       ccubind-resp ([ex] EQB toss ex arg envtail : ccexp-eq (BOD'' ex arg envtail)
                     (BOD' toss ex arg envtail)) ([n] cclets exh n) (EQB2 arg envtail toss exh))
 <- ({arg} {toss} {exh} {e}
       ccbind-resp ([envtail] EQB2 arg envtail toss exh) ([n] ccsnd e n) (EQB3 arg toss exh e))
 <- ({arg} {toss} {e}
       ccbind-resp ([exh] EQB3 arg toss exh e) ([n] ccfst e n) (EQB4 arg toss e))
 <- ({toss} {x}
       ccclosure-resp ([arg][e] EQB4 arg toss e) (EQE2 toss x) (EQ' toss x)).
- : vvpermavclosedvv  ([v] fzvv/closed) ([_][_] ccval-eq/).
- : vvpermavvclosedvv ([v] fzvv/closedvv) ([_][_] ccvval-eq/).
- : vvpermavvclosedvv ([v] fzvv/var) ([_][_] ccvval-eq/).

%worlds (blockw | blockccv | blockccvv) (vvpermavclosedvv _ _) (vvpermaclosedvv _ _)
                                        (vvpermavvclosedvv _ _).
%total (D E F) (vvpermaclosedvv D _) (vvpermavclosedvv E _) (vvpermavvclosedvv F _).

% now we need another |binders|^2 * |syntactic classes| lemmas, alas...

% freezing needs to preserve frozenness
permafrost : {ZN : {v:ccval} frozen ([y] N v y)}
             {FN : {y:ccval} freeze ([v] N v y) ([v] N' v y) (F y)}
             {ZN': {v:ccval} frozen ([y] N' v y)} type.
%mode permafrost +ZN +FN -ZN'.

permavfrost : {ZN : {v:ccval} vfrozen ([y] N v y)}
              {FN : {y:ccval} vfreeze ([v] N v y) ([v] N' v y) (F y)}
              {ZN': {v:ccval} vfrozen ([y] N' v y)} type.
%mode permavfrost +ZN +FN -ZN'.

permavvfrost : {ZN : {v:ccval} vvfrozen ([y] N v y)}
               {FN : {y:ccval} vvfreeze ([v] N v y) ([v] N' v y) (F y)}
               {ZN': {v:ccval} vvfrozen ([y] N' v y)} type.
%mode permavvfrost +ZN +FN -ZN'.

% freezing needs to preserve frozenvvness
vvpermafrost : {ZN : {v:ccval}  frozenvv ([y:ccvval] N v y)}
               {FN : {y:ccvval} freeze   ([v:ccval]  N v y) ([v:ccval] N' v y) (F y)}
               {ZN': {v:ccval}  frozenvv ([y:ccvval] N' v y)} type.
%mode vvpermafrost +ZN +FN -ZN'.

vvpermavfrost : {ZN : {v:ccval}  vfrozenvv ([y:ccvval] N y v)}
                {F2 : {y:ccvval} vfreeze   ([v:ccval] N y v) ([v:ccval] N' v y) (F y)}
                {ZN': {v:ccval}  vfrozenvv ([y:ccvval] N' v y)} type.
%mode vvpermavfrost +ZN +FN -ZN'.

vvpermavvfrost : {ZN : {v:ccval}  vvfrozenvv ([y:ccvval] N y v)}
                 {F2 : {y:ccvval} vvfreeze   ([v:ccval] N y v) ([v:ccval] N' v y) (F y)}
                 {ZN': {v:ccval}  vvfrozenvv ([y:ccvval] N' v y)} type.
%mode vvpermavvfrost +ZN +FN -ZN'.

- : vvpermafrost D ([_] fz/halt) D.
- : vvpermafrost ([x] fvv/call (Z1 x) (Z2 x)) ([yy] fz/call (F2 yy) (F1 yy))
                 ([x] fvv/call (Z1' x) (Z2' x))
 <- vvpermavfrost Z1 F1 Z1'
 <- vvpermavfrost Z2 F2 Z2'.
- : vvpermafrost ([x] fvv/fst (ZV x) (ZN x)) ([x] fz/fst (FN x) (FV x))
                 ([x] fvv/fst (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
 <- ({v} vvpermafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrost ([x] fvv/snd (ZV x) (ZN x)) ([x] fz/snd (FN x) (FV x))
                 ([x] fvv/snd (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
```

```
 <- ({v} vvpermafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrost ([x] fvv/put (ZV x) (ZN x)) ([x] fz/put (FN x) (FV x))
                 ([x] fvv/put (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
 <- ({vv} vvpermafrost ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : vvpermafrost ([x] fvv/lets (ZV x) (ZN x)) ([x] fz/lets (FN x) (FV x))
                 ([x] fvv/lets (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
 <- ({vv} vvpermafrost ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : vvpermafrost ([x] fvv/localhost (ZN x)) ([x] fz/localhost (FN x))
                 ([x] fvv/localhost (ZN' x))
 <- ({v} vvpermafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrost ([x] fvv/leta (ZV x) (ZN x)) ([x] fz/leta (FN x) (FV x))
                 ([x] fvv/leta (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
 <- ({v} vvpermafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrost ([x] fvv/unpack (ZV x) (ZN x)) ([x] fz/unpack (FN x) (FV x))
                 ([x] fvv/unpack (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
 <- ({w}{v} vvpermafrost ([x] ZN x w v) ([x] FN x w v) ([x] ZN' x w v)).
- : vvpermafrost ([x] fvv/wapp (ZV x) (ZN x)) ([x] fz/wapp (FN x) (FV x))
                 ([x] fvv/wapp (ZV' x) (ZN' x))
 <- vvpermavfrost ZV FV ZV'
 <- ({v} vvpermafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrost ([x] fvv/go (Z1 x) (Z2 x)) ([x] fz/go (F2 x) (F1 x))
                 ([x] fvv/go (Z1' x) (Z2' x))
 <- vvpermavfrost Z1 F1 Z1'
 <- vvpermavfrost Z2 F2 Z2'.
- : vvpermavfrost ([x] vfvv/ch (Z x)) ([x] fz/ch (FF x)) ([x] vfvv/ch (Z' x))
 <- vvpermavvfrost Z FF Z'.
- : vvpermavfrost ([x] vfvv/pair (Z2 x) (Z1 x)) ([x] fz/pair (F1 x) (F2 x))
                  ([x] vfvv/pair (Z2' x) (Z1' x))
 <- vvpermavfrost Z1 F1 Z1'
 <- vvpermavfrost Z2 F2 Z2'.
- : vvpermavfrost Z ([_] fz/closed) Z.
- : vvpermavfrost Z ([_] fz/1) Z.
- : vvpermavfrost ([_] vfvv/const) ([v] fz/const) ([_] vfvv/const).
- : vvpermavfrost ([v] vfvv/held (Z v)) ([v] fz/held (F v)) ([v] vfvv/held (Z' v))
 <- vvpermavfrost Z F Z'.
- : vvpermavfrost ([v] vfvv/wlam ([w:world] Z v w)) ([v] fz/wlam ([w] F v w))
                  ([v] vfvv/wlam ([w] Z' v w))
 <- ({w:world} vvpermavfrost ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : vvpermavfrost ([v] vfvv/pack (Z v)) ([v] fz/pack (F v)) ([v] vfvv/pack (Z' v))
 <- vvpermavfrost Z F Z'.
- : vvpermavfrost ([_] vfvv/closed) F Z'
 <- vvpermavclosed F EQ
 <- ({x} vfrozenvv-resp ([y] EQ y x) vfvv/closed (Z' x)).

- : vvpermavfrost ([v] vfvv/closure (Z v)) ([y:ccvval] fz/closure
                                              (F1 y : vfreeze ([v:ccval] V1 v y)
                                                              ([v:ccval] ENV' y v) (FZ y))
                                              ([a][e] F2 y a e)) ([v] Z'' v)
 <- vvpermavfrost Z F1 ZENV
 <- ({a}{e} vvpermaclosed ([v] F2 v a e) ([utoss:ccvval][x] EQ utoss x a e))
 <- ({toss:ccvval}{x:ccval}
       ccfu3-resp EQ
       ([f4]
          ccclosure ([arg][env]
                       ccfst env [exh]
                       ccsnd env [envtail]
                       ccleta exh [ex]
                       f4 toss ex arg envtail) (ccpair (ccheld x) (ENV' toss x)))
       (EQ2 toss x))
 <- ({x} vfrozenvv-resp ([y] EQ2 y x) (vfvv/closure (vfvv/pair vfvv/closed (ZENV x))) (Z'' x)).

- : vvpermavfrost ([v] vfvv/valid (Z v)) ([v] fz/valid (F v)) ([v] vfvv/valid (Z' v))
```

```
 <- vvpermavvfrost Z F Z'.

- : vvpermavvfrost ([v] vvfvv/vv ([w] Z v w)) ([v] fz/vv ([w] F v w))
                   ([v] vvfvv/vv ([w] Z' v w))
 <- ({w} vvpermavfrost ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : vvpermavvfrost Z ([_] fz/vclosed) Z.
- : vvpermavvfrost ([_] vvfvv/closed) F Z'
 <- vvpermavvclosed F EQ
 <- ({x} vvfrozenvv-resp ([y] EQ y x) vvfvv/closed (Z' x)).

%worlds (blockw | blockccv | blockccvv)
               (vvpermafrost _ _ _) (vvpermavfrost _ _ _) (vvpermavvfrost _ _ _).
%total (D E F) (vvpermafrost _ D _) (vvpermavfrost _ E _) (vvpermavvfrost _ F _).


%% another duplication
vvpermafrostvv :
      {ZN : {v:ccvval} frozenvv ([y:ccvval] N v y)}
      {FN : {y:ccvval} freezevv ([v:ccvval] N v y) ([v:ccvval] N' v y) (F y)}
      {ZN': {v:ccvval} frozenvv ([y:ccvval] N' v y)} type.
%mode vvpermafrostvv +ZN +FN -ZN'.

vvpermavfrostvv :
      {ZN : {v:ccvval} vfrozenvv ([y:ccvval] N y v)}
      {F2 : {y:ccvval} vfreezevv ([v:ccvval] N y v) ([v:ccvval] N' v y) (F y)}
      {ZN': {v:ccvval} vfrozenvv ([y:ccvval] N' v y)} type.
%mode vvpermavfrostvv +ZN +FN -ZN'.

vvpermavvfrostvv :
      {ZN : {v:ccvval} vvfrozenvv ([y:ccvval] N y v)}
      {F2 : {y:ccvval} vvfreezevv ([v:ccvval] N y v) ([v:ccvval] N' v y) (F y)}
      {ZN': {v:ccvval} vvfrozenvv ([y:ccvval] N' v y)} type.
%mode vvpermavvfrostvv +ZN +FN -ZN'.

- : vvpermafrostvv D ([_] fzvv/halt) D.
- : vvpermafrostvv ([x] fvv/call (Z1 x) (Z2 x)) ([yy] fzvv/call (F2 yy) (F1 yy))
                   ([x] fvv/call (Z1' x) (Z2' x))
 <- vvpermavfrostvv Z1 F1 Z1'
 <- vvpermavfrostvv Z2 F2 Z2'.
- : vvpermafrostvv ([x] fvv/fst (ZV x) (ZN x)) ([x] fzvv/fst (FN x) (FV x))
                   ([x] fvv/fst (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({v} vvpermafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrostvv ([x] fvv/snd (ZV x) (ZN x)) ([x] fzvv/snd (FN x) (FV x))
                   ([x] fvv/snd (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({v} vvpermafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrostvv ([x] fvv/put (ZV x) (ZN x)) ([x] fzvv/put (FN x) (FV x))
                   ([x] fvv/put (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({vv} vvpermafrostvv ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : vvpermafrostvv ([x] fvv/lets (ZV x) (ZN x)) ([x] fzvv/lets (FN x) (FV x))
                   ([x] fvv/lets (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({vv} vvpermafrostvv ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : vvpermafrostvv ([x] fvv/localhost (ZN x)) ([x] fzvv/localhost (FN x))
                   ([x] fvv/localhost (ZN' x))
 <- ({v} vvpermafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrostvv ([x] fvv/leta (ZV x) (ZN x)) ([x] fzvv/leta (FN x) (FV x))
                   ([x] fvv/leta (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({v} vvpermafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrostvv ([x] fvv/unpack (ZV x) (ZN x)) ([x] fzvv/unpack (FN x) (FV x))
                   ([x] fvv/unpack (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({w}{v} vvpermafrostvv ([x] ZN x w v) ([x] FN x w v) ([x] ZN' x w v)).
```

```
- : vvpermafrostvv ([x] fvv/wapp (ZV x) (ZN x)) ([x] fzvv/wapp (FN x) (FV x))
                   ([x] fvv/wapp (ZV' x) (ZN' x))
 <- vvpermavfrostvv ZV FV ZV'
 <- ({v} vvpermafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : vvpermafrostvv ([x] fvv/go (Z1 x) (Z2 x)) ([x] fzvv/go (F2 x) (F1 x))
                   ([x] fvv/go (Z1' x) (Z2' x))
 <- vvpermavfrostvv Z1 F1 Z1'
 <- vvpermavfrostvv Z2 F2 Z2'.
- : vvpermavfrostvv ([x] vfvv/ch (Z x)) ([x] fzvv/ch (FF x)) ([x] vfvv/ch (Z' x))
 <- vvpermavvfrostvv Z FF Z'.
- : vvpermavfrostvv ([x] vfvv/pair (Z2 x) (Z1 x)) ([x] fzvv/pair (F1 x) (F2 x))
                    ([x] vfvv/pair (Z2' x) (Z1' x))
 <- vvpermavfrostvv Z1 F1 Z1'
 <- vvpermavfrostvv Z2 F2 Z2'.
- : vvpermavfrostvv Z ([_] fzvv/closed) Z.
- : vvpermavfrostvv Z ([_] fzvv/1) Z.
- : vvpermavfrostvv ([_] vfvv/const) ([v] fzvv/const) ([_] vfvv/const).
- : vvpermavfrostvv ([v] vfvv/held (Z v)) ([v] fzvv/held (F v)) ([v] vfvv/held (Z' v))
 <- vvpermavfrostvv Z F Z'.
- : vvpermavfrostvv ([v] vfvv/wlam ([w:world] Z v w)) ([v] fzvv/wlam ([w] F v w))
                    ([v] vfvv/wlam ([w] Z' v w))
 <- ({w:world} vvpermavfrostvv ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : vvpermavfrostvv ([v] vfvv/pack (Z v)) ([v] fzvv/pack (F v)) ([v] vfvv/pack (Z' v))
 <- vvpermavfrostvv Z F Z'.
- : vvpermavfrostvv ([_] vfvv/closed) F Z'
 <- vvpermavclosedvv F EQ
 <- ({x} vfrozenvv-resp ([y] EQ y x) vfvv/closed (Z' x)).
- : vvpermavfrostvv ([v] vfvv/closure (Z v)) ([y:ccvval] fzvv/closure
                                               (F1 y : vfreezevv ([v:ccvval] V1 v y)
                                                                 ([v:ccvval] ENV' y v) (FZ y))
                                               ([a][e] F2 y a e)) ([v] Z'' v)
 <- vvpermavfrostvv Z F1 ZENV
 <- ({a}{e} vvpermaclosedvv ([v] F2 v a e) ([utoss:ccvval][x] EQ utoss x a e))
 <- ({toss:ccvval}{x:ccvval}
       ccfuu2-resp EQ
       ([f4]
          ccclosure ([arg][env]
                       ccfst env [exh]
                       ccsnd env [envtail]
                       cclets exh [ex]
                       f4 toss ex arg envtail) (ccpair (ccsh x) (ENV' toss x)))
       (EQ2 toss x))
 <- ({x} vfrozenvv-resp ([y] EQ2 y x)
          (vfvv/closure (vfvv/pair vfvv/closed (ZENV x))) (Z'' x)).

- : vvpermavfrostvv ([v] vfvv/valid (Z v)) ([v] fzvv/valid (F v)) ([v] vfvv/valid (Z' v))
 <- vvpermavvfrostvv Z F Z'.
- : vvpermavvfrostvv ([v] vvfvv/vv ([w] Z v w)) ([v] fzvv/vv ([w] F v w))
                     ([v] vvfvv/vv ([w] Z' v w))
 <- ({w} vvpermavfrostvv ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : vvpermavvfrostvv Z ([_] fzvv/closedvv) Z.
- : vvpermavvfrostvv ([_] vvfvv/closed) F Z'
 <- vvpermavvclosedvv F EQ
 <- ({x} vvfrozenvv-resp ([y] EQ y x) vvfvv/closed (Z' x)).

%worlds (blockw | blockccv | blockccvv)
    (vvpermafrostvv _ _ _) (vvpermavfrostvv _ _ _) (vvpermavvfrostvv _ _ _).

%total (D E F) (vvpermafrostvv _ D _) (vvpermavfrostvv _ E _) (vvpermavvfrostvv _ F _).

%% end duplication

- : permafrost D ([_] fz/halt) D.
- : permafrost ([x] f/call (Z1 x) (Z2 x)) ([x] fz/call (F2 x) (F1 x))
               ([x] f/call (Z1' x) (Z2' x))
     <- permavfrost Z1 F1 Z1'
```

```
     <- permavfrost Z2 F2 Z2'.
- : permafrost ([x] f/put (ZV x) (ZN x)) ([x] fz/put (FN x) (FV x))
               ([x] f/put (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({vv} permafrost ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : permafrost ([x] f/lets (ZV x) (ZN x)) ([x] fz/lets (FN x) (FV x))
               ([x] f/lets (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({vv} permafrost ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : permafrost ([x] f/localhost (ZN x)) ([x] fz/localhost (FN x))
               ([x] f/localhost (ZN' x))
     <- ({v} permafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrost ([x] f/leta (ZV x) (ZN x)) ([x] fz/leta (FN x) (FV x))
               ([x] f/leta (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({v} permafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrost ([x] f/fst (ZV x) (ZN x)) ([x] fz/fst (FN x) (FV x))
               ([x] f/fst (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({v} permafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrost ([x] f/snd (ZV x) (ZN x)) ([x] fz/snd (FN x) (FV x))
               ([x] f/snd (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({v} permafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrost ([x] f/unpack (ZV x) (ZN x)) ([x] fz/unpack (FN x) (FV x))
               ([x] f/unpack (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({w}{v} permafrost ([x] ZN x w v) ([x] FN x w v) ([x] ZN' x w v)).
- : permafrost ([x] f/wapp (ZV x) (ZN x)) ([x] fz/wapp (FN x) (FV x))
               ([x] f/wapp (ZV' x) (ZN' x))
     <- permavfrost ZV FV ZV'
     <- ({v} permafrost ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrost ([x] f/go (Z1 x) (Z2 x)) ([x] fz/go (F2 x) (F1 x))
               ([x] f/go (Z1' x) (Z2' x))
     <- permavfrost Z1 F1 Z1'
     <- permavfrost Z2 F2 Z2'.
- : permavfrost ([x] vf/ch (Z x)) ([x] fz/ch (FF x)) ([x] vf/ch (Z' x))
     <- permavvfrost Z FF Z'.
- : permavfrost ([x] vf/pair (Z2 x) (Z1 x)) ([x] fz/pair (F1 x) (F2 x))
                ([x] vf/pair (Z2' x) (Z1' x))
     <- permavfrost Z1 F1 Z1'
     <- permavfrost Z2 F2 Z2'.
- : permafrost ([_] f/closed) F Z'
 <- permaclosed F EQ
 <- ({x} frozen-resp ([y] EQ y x) f/closed (Z' x)).
- : permavfrost ([_] vf/closed) F Z'
 <- permavclosed F EQ
 <- ({x} vfrozen-resp ([y] EQ y x) vf/closed (Z' x)).
- : permavfrost Z ([_] fz/closed) Z.
- : permavfrost Z ([_] fz/1) Z.
- : permavfrost ([_] vf/const) ([v] fz/const) ([_] vf/const).
- : permavfrost ([v] vf/held (Z v)) ([v] fz/held (F v)) ([v] vf/held (Z' v))
 <- permavfrost Z F Z'.
- : permavfrost ([v] vf/wlam ([w:world] Z v w)) ([v] fz/wlam ([w] F v w))
                ([v] vf/wlam ([w] Z' v w))
 <- ({w:world} permavfrost ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : permavfrost ([v] vf/pack (Z v)) ([v] fz/pack (F v)) ([v] vf/pack (Z' v))
 <- permavfrost Z F Z'.
- : permavfrost ([v] vf/closure (Z v)) ([v] fz/closure (F1 v) ([a][e] F2 v a e)) ([v] Z'' v)
 <- permavfrost Z F1 ZENV
 <- ({a}{e} permaclosed ([v] F2 v a e) ([toss][x] EQ x toss a e))
 <- ({toss:ccval}{x}
       ccf4-resp EQ
       ([f4]
          ccclosure ([arg][env]
                       ccfst env [exh]
```

```
                    ccsnd env [envtail]
                    ccleta exh [ex]
                    f4 ex toss arg envtail) (ccpair (ccheld x) (ENV' toss x)))
      (EQ2 toss x))
 <- ({x} vfrozen-resp ([y] EQ2 y x) (vf/closure (vf/pair vf/closed (ZENV x))) (Z'' x)).
- : permavfrost ([v] vf/valid (Z v)) ([v] fz/valid (F v)) ([v] vf/valid (Z' v))
 <- permavvfrost Z F Z'.
- : permavvfrost ([v] vvf/vv ([w] Z v w)) ([v] fz/vv ([w] F v w)) ([v] vvf/vv ([w] Z' v w))
 <- ({w} permavfrost ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : permavvfrost Z ([_] fz/vclosed) Z.
- : permavvfrost ([_] vvf/closed) F Z'
 <- permavvclosed F EQ
 <- ({x} vvfrozen-resp ([y] EQ y x) vvf/closed (Z' x)).

%worlds (blockw | blockccv | blockccvv)
              (permafrost _ _ _) (permavfrost _ _ _) (permavvfrost _ _ _).
%total (D E F) (permafrost _ D _) (permavfrost _ E _) (permavvfrost _ F _).

permafrostvv :
      {ZN : {u:ccvval}  frozen   ([v:ccval] N u v)}
      {FN : {v:ccval}   freezevv ([u:ccvval] N u v) ([u:ccvval] N' u v) (F v)}
      {ZN': {u:ccvval}  frozen   ([v:ccval] N' u v)} type.
%mode permafrostvv +ZN +FN -ZN'.

permavfrostvv :
      {ZN : {u:ccvval}  vfrozen   ([v:ccval] N u v)}
      {FN : {v:ccval}   vfreezevv ([u:ccvval] N u v) ([u:ccvval] N' u v) (F v)}
      {ZN': {u:ccvval}  vfrozen   ([v:ccval] N' u v)} type.
%mode permavfrostvv +ZN +FN -ZN'.

permavvfrostvv :
      {ZN : {u:ccvval}  vvfrozen   ([v:ccval] N u v)}
      {FN : {v:ccval}   vvfreezevv ([u:ccvval] N u v) ([u:ccvval] N' u v) (F v)}
      {ZN': {u:ccvval}  vvfrozen   ([v:ccval] N' u v)} type.
%mode permavvfrostvv +ZN +FN -ZN'.


%% duplicated from above
- : permafrostvv D ([_] fzvv/halt) D.
- : permafrostvv ([x] f/call (Z1 x) (Z2 x)) ([x] fzvv/call (F2 x) (F1 x))
                 ([x] f/call (Z1' x) (Z2' x))
     <- permavfrostvv Z1 F1 Z1'
     <- permavfrostvv Z2 F2 Z2'.
- : permafrostvv ([x] f/put (ZV x) (ZN x)) ([x] fzvv/put (FN x) (FV x))
                 ([x] f/put (ZV' x) (ZN' x))
     <- permavfrostvv ZV FV ZV'
     <- ({vv} permafrostvv ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : permafrostvv ([x] f/lets (ZV x) (ZN x)) ([x] fzvv/lets (FN x) (FV x))
                 ([x] f/lets (ZV' x) (ZN' x))
     <- permavfrostvv ZV FV ZV'
     <- ({vv} permafrostvv ([x] ZN x vv) ([x] FN x vv) ([x] ZN' x vv)).
- : permafrostvv ([x] f/localhost (ZN x)) ([x] fzvv/localhost (FN x))
                 ([x] f/localhost (ZN' x))
     <- ({v} permafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrostvv ([x] f/leta (ZV x) (ZN x)) ([x] fzvv/leta (FN x) (FV x))
                 ([x] f/leta (ZV' x) (ZN' x))
     <- permavfrostvv ZV FV ZV'
     <- ({v} permafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrostvv ([x] f/fst (ZV x) (ZN x)) ([x] fzvv/fst (FN x) (FV x))
                 ([x] f/fst (ZV' x) (ZN' x))
     <- permavfrostvv ZV FV ZV'
     <- ({v} permafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrostvv ([x] f/snd (ZV x) (ZN x)) ([x] fzvv/snd (FN x) (FV x))
                 ([x] f/snd (ZV' x) (ZN' x))
     <- permavfrostvv ZV FV ZV'
     <- ({v} permafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
```

```
- : permafrostvv ([x] f/unpack (ZV x) (ZN x)) ([x] fzvv/unpack (FN x) (FV x))
                 ([x] f/unpack (ZV' x) (ZN' x))
    <- permavfrostvv ZV FV ZV'
    <- ({w}{v} permafrostvv ([x] ZN x w v) ([x] FN x w v) ([x] ZN' x w v)).
- : permafrostvv ([x] f/wapp (ZV x) (ZN x)) ([x] fzvv/wapp (FN x) (FV x))
                 ([x] f/wapp (ZV' x) (ZN' x))
    <- permavfrostvv ZV FV ZV'
    <- ({v} permafrostvv ([x] ZN x v) ([x] FN x v) ([x] ZN' x v)).
- : permafrostvv ([x] f/go (Z1 x) (Z2 x)) ([x] fzvv/go (F2 x) (F1 x))
                 ([x] f/go (Z1' x) (Z2' x))
    <- permavfrostvv Z1 F1 Z1'
    <- permavfrostvv Z2 F2 Z2'.
- : permavfrostvv ([x] vf/ch (Z x)) ([x] fzvv/ch (FF x)) ([x] vf/ch (Z' x))
    <- permavvfrostvv Z FF Z'.
- : permavfrostvv ([x] vf/pair (Z2 x) (Z1 x)) ([x] fzvv/pair (F1 x) (F2 x))
                  ([x] vf/pair (Z2' x) (Z1' x))
    <- permavfrostvv Z1 F1 Z1'
    <- permavfrostvv Z2 F2 Z2'.
- : permafrostvv ([_] f/closed) F Z'
 <- permaclosedvv F EQ
 <- ({x} frozen-resp ([y] EQ y x) f/closed (Z' x)).
- : permavfrostvv ([_] vf/closed) F Z'
 <- permavclosedvv F EQ
 <- ({x} vfrozen-resp ([y] EQ y x) vf/closed (Z' x)).
- : permavfrostvv Z ([_] fzvv/closed) Z.
- : permavfrostvv Z ([_] fzvv/1) Z.
- : permavfrostvv ([_] vf/const) ([v] fzvv/const) ([_] vf/const).
- : permavfrostvv ([v] vf/held (Z v)) ([v] fzvv/held (F v)) ([v] vf/held (Z' v))
 <- permavfrostvv Z F Z'.
- : permavfrostvv ([v] vf/wlam ([w:world] Z v w)) ([v] fzvv/wlam ([w] F v w))
                  ([v] vf/wlam ([w] Z' v w))
 <- ({w:world} permavfrostvv ([v] Z v w) ([v] F v w) ([v] Z' v w)).
- : permavfrostvv ([v] vf/pack (Z v)) ([v] fzvv/pack (F v)) ([v] vf/pack (Z' v))
 <- permavfrostvv Z F Z'.
- : permavfrostvv ([v] vf/valid (Z v)) ([v] fzvv/valid (F v)) ([v] vf/valid (Z' v))
 <- permavvfrostvv Z F Z'.
- : permavvfrostvv ([v] vvf/vv ([w] Z v w)) ([v] fzvv/vv ([w] F v w))
                   ([v] vvf/vv ([w] Z' v w))
 <- ({w} permavfrostvv ([v] Z v w) ([v] F v w) ([v] Z' v w)).
%%% end duplicated

- : permavfrostvv ([v] vf/closure (Z v)) ([v] fzvv/closure (F1 v) ([a][e] F2 v a e))
                  ([v] Z'' v)
 <- permavfrostvv Z F1 ZENV
 <- ({a}{e} permaclosedvv ([v] F2 v a e) ([toss][x] EQ x toss a e))
 <- ({toss:ccval}{x}
       ccfu3-resp EQ
       ([f4]
          ccclosure ([arg][env]
                       ccfst env [exh]
                       ccsnd env [envtail]
                       cclets exh [ex]
                       f4 ex toss arg envtail) (ccpair (ccsh x) (ENV' toss x)))
       (EQ2 toss x))
 <- ({x} vfrozen-resp ([y] EQ2 y x) (vf/closure (vf/pair vf/closed (ZENV x))) (Z'' x)).

- : permavvfrostvv Z ([_] fzvv/closedvv) Z.
- : permavvfrostvv ([_] vvf/closed) F Z'
 <- permavvclosedvv F EQ
 <- ({x} vvfrozen-resp ([y] EQ y x) vvf/closed (Z' x)).

%worlds (blockw | blockccv | blockccvv)
               (permafrostvv _ _ _) (permavfrostvv _ _ _) (permavvfrostvv _ _ _).
%total (D E F) (permafrostvv _ D _) (permavfrostvv _ E _) (permavvfrostvv _ F _).
```

```
% freezing needs to preserve well-formedness.

% freeze a regular variable within an expression
freeze/ok : {WN  : {x}{xok:ccofv x A W} ccof (N x) W'}
            {D   : freeze N N' F}
            {WN' : {x}{xok:ccofv x A W} ccof (N' x) W'}
            type.
%mode freeze/ok +WN +D -WN'.

vfreeze/ok : {WV  : {x}{xok:ccofv x A W} ccofv (V x) B W'}
             {D   : vfreeze V V' F}
             {WV' : {x}{xok:ccofv x A W} ccofv (V' x) B W'}
             type.
%mode vfreeze/ok +WV +D -WV'.

vvfreeze/ok : {WVV  : {x}{xok:ccofv x A W} ccofvv (VV x) B}
              {D    : vvfreeze VV VV' F}
              {WVV' : {x}{xok:ccofv x A W} ccofvv (VV' x) B}
              type.
%mode vvfreeze/ok +WVV +D -WVV'.

- : vfreeze/ok D fz/var D.
- : vfreeze/ok D fz/closed D.
- : freeze/ok ([_][_] cco_halt) fz/halt ([_][_] cco_halt).
- : freeze/ok ([x][wx] cco_call (WF x wx) (WA x wx)) (fz/call FA FF)
              ([x][wx] cco_call (WF' x wx) (WA' x wx))
 <- vfreeze/ok WF FF WF'
 <- vfreeze/ok WA FA WA'.
- : freeze/ok ([x][wx] cco_go (WF x wx) (WA x wx)) (fz/go FA FF)
              ([x][wx] cco_go (WF' x wx) (WA' x wx))
 <- vfreeze/ok WF FF WF'
 <- vfreeze/ok WA FA WA'.
- : freeze/ok ([x][wx] cco_fst (WV x wx) ([y][wy] WN y wy x wx) (ZN x)) (fz/fst FN FV)
              ([x][wx] cco_fst (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreeze/ok WV FV WV'
 <- ({x}{wx} freeze/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrost ZN FN ZN'.
- : freeze/ok ([x][wx] cco_snd (WV x wx) ([y][wy] WN y wy x wx) (ZN x)) (fz/snd FN FV)
              ([x][wx] cco_snd (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreeze/ok WV FV WV'
 <- ({x}{wx} freeze/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrost ZN FN ZN'.
- : freeze/ok
        ([x][wx] cco_put MOB (WV x wx) ([yy][wyy:ccofvv yy ([_] A)] WN yy wyy x wx) (ZN x))
        (fz/put FN FV) ([x][wx] cco_put MOB (WV' x wx) ([yy][wyy] WN' yy wyy x wx) (ZN' x))
 <- vfreeze/ok WV FV WV'
 <- ({xx}{wxx} freeze/ok ([y][wy] WN xx wxx y wy) (FN xx) ([y][wy] WN' xx wxx y wy))
 <- vvpermafrost ZN FN ZN'.
- : freeze/ok ([x][wx] cco_lets (WV x wx) ([yy][wyy:ccofvv yy A] WN yy wyy x wx) (ZN x))
          (fz/lets FN FV) ([x][wx] cco_lets (WV' x wx) ([yy][wyy] WN' yy wyy x wx) (ZN' x))
 <- vfreeze/ok WV FV WV'
 <- ({xx}{wxx} freeze/ok ([y][wy] WN xx wxx y wy) (FN xx) ([y][wy] WN' xx wxx y wy))
 <- vvpermafrost ZN FN ZN'.
- : freeze/ok ([x][wx] cco_localhost ([y][wy] WN y wy x wx) (ZN x)) (fz/localhost FN)
              ([x][wx] cco_localhost ([y][wy] WN' y wy x wx) (ZN' x))
 <- ({x}{wx} freeze/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrost ZN FN ZN'.
- : freeze/ok ([x][wx] cco_leta (WV x wx) ([y][wy] WN y wy x wx) (ZN x)) (fz/leta FN FV)
              ([x][wx] cco_leta (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreeze/ok WV FV WV'
 <- ({x}{wx} freeze/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrost ZN FN ZN'.
- : freeze/ok
    ([x][wx] cco_unpack (WV x wx) ([w][y][wy] WN w y wy x wx) ([w] ZN w x)) (fz/unpack FN FV)
    ([x][wx] cco_unpack (WV' x wx) ([w][y][wy] WN' w y wy x wx) ([w] ZN' w x))
 <- vfreeze/ok WV FV WV'
```

```
 <- ({w}{x}{wx} freeze/ok (WN w x wx) (FN w x) (WN' w x wx))
 <- ({w} permafrost (ZN w) (FN w) (ZN' w)).
- : freeze/ok ([x][wx] cco_wapp (WV x wx : ccofv _ (call A) _)
                       ([y][wy : ccofv y (A W') W] WN y wy x wx) (ZN x)) (fz/wapp FN FV)
              ([x][wx] cco_wapp (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreeze/ok WV FV WV'
 <- ({x}{wx} freeze/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrost ZN FN ZN'.
- : vfreeze/ok ([x][wx] ccov_pair (W1 x wx) (W2 x wx)) (fz/pair F2 F1)
               ([x][wx] ccov_pair (W1' x wx) (W2' x wx))
 <- vfreeze/ok W1 F1 W1'
 <- vfreeze/ok W2 F2 W2'.
- : vfreeze/ok _ fz/1 ([x][wx] ccov_unit).
- : vfreeze/ok ([x][wx] ccov_ch (WVV x wx)) (fz/ch F) ([x][wx] ccov_ch (WVV' x wx))
 <- vvfreeze/ok WVV F WVV'.
- : vfreeze/ok
    ([x][wx:ccofv x AFREE W']
       ccov_closure (WENV x wx : ccofv (ENV x) AENV W)
                    ([a][wa : ccofv a AARG W]
                     [e][we : ccofv e AENV W]
                       WBOD x wx a wa e we) (ZwrtENV x : {a} frozen ([env] BOD x a env))
                                            (ZwrtARG x : {e} frozen ([arg] BOD x arg e)))
    (fz/closure FENV ([a][e] FBOD a e : freeze _ ([x] BOD' x a e) (ZBOD a e)))
%%
    ([x][wx:ccofv x AFREE W']
       ccov_closure (ccov_pair (ccov_held wx) (WENV' x wx))
                    ([a][wa : ccofv a AARG W]
                     [e][we : ccofv e ((AFREE cat W') c& AENV) W]
                       cco_fst we
                       ([exh][wexh:ccofv exh (AFREE cat W') W]
                          cco_snd we
                          ([envtail][wenvtail:ccofv envtail AENV W]
                             cco_leta wexh
                             ([ex][wex:ccofv ex AFREE W']
                                WBOD' ex wex a wa envtail wenvtail
                                ) % leta body
                             (ZBOD a envtail) % frozen wrt leta
                             ) % snd body
                          (f/leta vf/closed
                             [y] ZwrtENV' y a) % frozen wrt snd proj
                          ) % fst body
                       (f/snd vf/closed [envtail]
                          f/leta vf/var [ex]
                          f/closed
                          ) %{ frozen wrt fst proj }% )
                    ([v]
                       f/fst vf/var [exh]
                       f/snd vf/var [envtail]
                       f/leta vf/closed [ex]
                       f/closed)
                    ([v]
                       f/fst vf/closed [exh]
                       f/snd vf/closed [envtail]
                       f/leta vf/closed [ex]
                       ZwrtARG' ex envtail))
%%
 <- vfreeze/ok WENV FENV (WENV' : {x}{ofx} ccofv (ENV' x) AENV W)
 <- ({a}{wa}{e}{we}
       freeze/ok ([x][wx] WBOD x wx a wa e we) (FBOD a e)
                 ([x][wx] WBOD' x wx a wa e we : ccof (BOD' x a e) W))
 <- ({z} permafrost ([x] ZwrtARG x z) ([y] FBOD y z)
         ([x] ZwrtARG' x z : frozen ([y] BOD' x y z)))
 <- ({y} permafrost ([x] ZwrtENV x y) ([z] FBOD y z)
         ([x] ZwrtENV' x y : frozen ([z] BOD' x y z))).

- : vfreeze/ok ([x][wx] ccov_pack AF (W x wx)) (fz/pack F) ([x][wx] ccov_pack AF (W' x wx))
```

```
 <- vfreeze/ok W F W'.
- : vfreeze/ok ([x][wx] ccov_wlam ([w] WV x wx w)) (fz/wlam ([w] F w))
               ([x][wx] ccov_wlam ([w] WV' x wx w))
 <- ({w} vfreeze/ok ([x][wx] WV x wx w) (F w) ([x][wx] WV' x wx w)).
- : vfreeze/ok D fz/const D.
- : vfreeze/ok ([x][wx] ccov_held (W x wx)) (fz/held F) ([x][wx] ccov_held (W' x wx))
 <- vfreeze/ok W F W'.
- : vfreeze/ok ([x:ccval][wx:ccofv x A W] ccov_valid (WVV x wx : ccofvv _ Af))
               (fz/valid F) ([x][wx] ccov_valid (WVV' x wx))
 <- vvfreeze/ok WVV F WVV'.
- : vvfreeze/ok ([x][wx] ccovv ([w] WV x wx w)) (fz/vv ([w] F w))
                ([x][wx] ccovv ([w] WV' x wx w))
 <- ({w} vfreeze/ok ([x][wx] WV x wx w) (F w) ([x][wx] WV' x wx w)).
- : vvfreeze/ok D fz/vclosed D.

%block blockccvok  : some {W:world} {A:ctyp} block {x:ccval}{xok:ccofv x A W}.
%block blockccvvok : some {Af:world -> ctyp} block {xx:ccvval}{xxok:ccofvv xx Af}.

%worlds (blockw | blockccvok | blockccvvok)
               (freeze/ok _ _ _) (vfreeze/ok _ _ _) (vvfreeze/ok _ _ _).
%total (D E F) (freeze/ok _ D _) (vfreeze/ok _ E _) (vvfreeze/ok _ F _).

% freezevv a valid variable within an expression
freezevv/ok : {WN  : {u}{uok:ccofvv u A} ccof (N u) W'}
              {D   : freezevv N N' F}
              {WN' : {u}{uok:ccofvv u A} ccof (N' u) W'} type.
%mode freezevv/ok +WN +D -WN'.

vfreezevv/ok : {WV  : {u}{uok:ccofvv u A} ccofv (V u) B W'}
               {D   : vfreezevv V V' F}
               {WV' : {u}{uok:ccofvv u A} ccofv (V' u) B W'} type.
%mode vfreezevv/ok +WV +D -WV'.

vvfreezevv/ok : {WVV  : {u}{uok:ccofvv u A} ccofvv (VV u) B}
                {D    : vvfreezevv VV VV' F}
                {WVV' : {u}{uok:ccofvv u A} ccofvv (VV' u) B} type.
%mode vvfreezevv/ok +WVV +D -WVV'.

- : freezevv/ok ([x][wx] cco_go (WF x wx) (WA x wx)) (fzvv/go FA FF)
                ([x][wx] cco_go (WF' x wx) (WA' x wx))
 <- vfreezevv/ok WF FF WF'
 <- vfreezevv/ok WA FA WA'.
- : freezevv/ok ([_][_] cco_halt) fzvv/halt ([_][_] cco_halt).
- : freezevv/ok ([x][wx] cco_call (WF x wx) (WA x wx)) (fzvv/call FA FF)
                ([x][wx] cco_call (WF' x wx) (WA' x wx))
 <- vfreezevv/ok WF FF WF'
 <- vfreezevv/ok WA FA WA'.
- : freezevv/ok ([x][wx] cco_fst (WV x wx) ([y][wy] WN y wy x wx) (ZN x)) (fzvv/fst FN FV)
                ([x][wx] cco_fst (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreezevv/ok WV FV WV'
 <- ({x}{wx} freezevv/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrostvv ZN FN ZN'.
- : freezevv/ok ([x][wx] cco_snd (WV x wx) ([y][wy] WN y wy x wx) (ZN x)) (fzvv/snd FN FV)
                ([x][wx] cco_snd (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreezevv/ok WV FV WV'
 <- ({x}{wx} freezevv/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrostvv ZN FN ZN'.
- : freezevv/ok
       ([x][wx] cco_put MOB (WV x wx) ([yy][wyy:ccofvv yy ([w] A)] WN yy wyy x wx) (ZN x))
       (fzvv/put FN FV) ([x][wx] cco_put MOB (WV' x wx) ([yy][wyy] WN' yy wyy x wx) (ZN' x))
 <- vfreezevv/ok WV FV WV'
 <- ({xx}{wxx} freezevv/ok ([y][wy] WN xx wxx y wy) (FN xx) ([y][wy] WN' xx wxx y wy))
 <- vvpermafrostvv ZN FN ZN'.
- : freezevv/ok ([x][wx] cco_lets (WV x wx) ([yy][wyy:ccofvv yy A] WN yy wyy x wx) (ZN x))
      (fzvv/lets FN FV) ([x][wx] cco_lets (WV' x wx) ([yy][wyy] WN' yy wyy x wx) (ZN' x))
 <- vfreezevv/ok WV FV WV'
```

```
 <- ({xx}{wxx} freezevv/ok ([y][wy] WN xx wxx y wy) (FN xx) ([y][wy] WN' xx wxx y wy))
 <- vvpermafrostvv ZN FN ZN'.
- : freezevv/ok ([x][wx] cco_localhost ([y][wy] WN y wy x wx) (ZN x)) (fzvv/localhost FN)
                ([x][wx] cco_localhost ([y][wy] WN' y wy x wx) (ZN' x))
 <- ({x}{wx} freezevv/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrostvv ZN FN ZN'.
- : freezevv/ok ([x][wx] cco_leta (WV x wx) ([y][wy] WN y wy x wx) (ZN x)) (fzvv/leta FN FV)
                ([x][wx] cco_leta (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreezevv/ok WV FV WV'
 <- ({x}{wx} freezevv/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrostvv ZN FN ZN'.
- : freezevv/ok ([x][wx] cco_unpack (WV x wx) ([w][y][wy] WN w y wy x wx) ([w] ZN w x))
                (fzvv/unpack FN FV)
                ([x][wx] cco_unpack (WV' x wx) ([w][y][wy] WN' w y wy x wx) ([w] ZN' w x))
 <- vfreezevv/ok WV FV WV'
 <- ({w}{x}{wx} freezevv/ok (WN w x wx) (FN w x) (WN' w x wx))
 <- ({w} permafrostvv (ZN w) (FN w) (ZN' w)).
- : freezevv/ok ([x][wx] cco_wapp (WV x wx : ccofv _ (call A) _)
                                 ([y][wy : ccofv y (A W') W] WN y wy x wx) (ZN x))
          (fzvv/wapp FN FV) ([x][wx] cco_wapp (WV' x wx) ([y][wy] WN' y wy x wx) (ZN' x))
 <- vfreezevv/ok WV FV WV'
 <- ({x}{wx} freezevv/ok (WN x wx) (FN x) (WN' x wx))
 <- permafrostvv ZN FN ZN'.
- : vfreezevv/ok ([x][wx] ccov_pair (W1 x wx) (W2 x wx)) (fzvv/pair F2 F1)
                 ([x][wx] ccov_pair (W1' x wx) (W2' x wx))
 <- vfreezevv/ok W1 F1 W1'
 <- vfreezevv/ok W2 F2 W2'.
- : vfreezevv/ok _ fzvv/1 ([x][wx] ccov_unit).
- : vfreezevv/ok ([x][wx] ccov_ch (WVV x wx)) (fzvv/ch F) ([x][wx] ccov_ch (WVV' x wx))
 <- vvfreezevv/ok WVV F WVV'.
- : vfreezevv/ok ([x][wx] ccov_pack AF (W x wx)) (fzvv/pack F)
                 ([x][wx] ccov_pack AF (W' x wx))
 <- vfreezevv/ok W F W'.
- : vfreezevv/ok ([x][wx] ccov_wlam ([w] WV x wx w)) (fzvv/wlam ([w] F w))
                 ([x][wx] ccov_wlam ([w] WV' x wx w))
 <- ({w} vfreezevv/ok ([x][wx] WV x wx w) (F w) ([x][wx] WV' x wx w)).
- : vfreezevv/ok D fzvv/const D.
- : vfreezevv/ok ([x][wx] ccov_held (W x wx)) (fzvv/held F) ([x][wx] ccov_held (W' x wx))
 <- vfreezevv/ok W F W'.
- : vfreezevv/ok ([x:ccvval][wx:ccofvv x Af] ccov_valid (WVV x wx : ccofvv _ Bf))
                 (fzvv/valid F) ([x][wx] ccov_valid (WVV' x wx))
 <- vvfreezevv/ok WVV F WVV'.
- : vvfreezevv/ok ([x][wx] ccovv ([w] WV x wx w)) (fzvv/vv ([w] F w))
                  ([x][wx] ccovv ([w] WV' x wx w))
 <- ({w} vfreezevv/ok ([x][wx] WV x wx w) (F w) ([x][wx] WV' x wx w)).
- : vfreezevv/ok D fzvv/closed D.
- : vfreezevv/ok
       ([x][wx:ccofvv x AFREE]
          ccov_closure (WENV x wx : ccofv (ENV x) AENV W)
                       ([a][wa : ccofv a AARG W]
                        [e][we : ccofv e AENV W]
                          WBOD x wx a wa e we) (ZwrtENV x : {a} frozen ([env] BOD x a env))
                                               (ZwrtARG x : {e} frozen ([arg] BOD x arg e)))
       (fzvv/closure FENV ([a][e] FBOD a e : freezevv _ ([x] BOD' x a e) (ZBOD a e)))
%%
       ([x][wx:ccofvv x AFREE]
          ccov_closure (ccov_pair (ccov_ch wx) (WENV' x wx))
                       ([a][wa : ccofv a AARG W]
                        [e][we : ccofv e ((csh AFREE) c& AENV) W]
                          cco_fst we
                          ([exh][wexh:ccofv exh (csh AFREE) W]
                             cco_snd we
                             ([envtail][wenvtail:ccofv envtail AENV W]
                                cco_lets wexh
                                ([ex][wex:ccofvv ex AFREE]
                                   WBOD' ex wex a wa envtail wenvtail
```

```
                                    ) % lets body
                                  (ZBOD a envtail) % frozen wrt lets
                                  ) % snd body
                               (f/lets vf/closed
                                  [y] ZwrtENV' y a) % frozen wrt snd proj
                               ) % fst body
                            (f/snd vf/closed [envtail]
                               f/lets vf/var [ex]
                               f/closed
                               ) %{ frozen wrt fst proj }% )
                         ([v]
                            f/fst vf/var [exh]
                            f/snd vf/var [envtail]
                            f/lets vf/closed [ex]
                            f/closed)
                         ([v]
                            f/fst vf/closed [exh]
                            f/snd vf/closed [envtail]
                            f/lets vf/closed [ex]
                            ZwrtARG' ex envtail))
%%
 <- vfreezevv/ok WENV FENV (WENV' : {x}{ofx} ccofv (ENV' x) AENV W)
 <- ({a}{wa}{e}{we}
       freezevv/ok ([x][wx] WBOD x wx a wa e we) (FBOD a e)
                   ([x][wx] WBOD' x wx a wa e we : ccof (BOD' x a e) W))
   <- ({z}
         permafrostvv ([x] ZwrtARG x z) ([y] FBOD y z)
         ([x] ZwrtARG' x z : frozen ([y] BOD' x y z)))
   <- ({y}
         permafrostvv ([x] ZwrtENV x y) ([z] FBOD y z)
         ([x] ZwrtENV' x y : frozen ([z] BOD' x y z))).

- : vvfreezevv/ok D fzvv/closedvv D.
- : vvfreezevv/ok D fzvv/var D.

%block blockccvok  : some {W:world} {A:ctyp} block {x: ccval} {xok: ccofv x A W}.
%block blockccvvok : some {Af:world -> ctyp} block {xx:ccvval}{xxok:ccofvv xx Af}.

%worlds (blockw | blockccvok | blockccvvok)
               (freezevv/ok _ _ _) (vfreezevv/ok _ _ _) (vvfreezevv/ok _ _ _).
%total (D E F) (freezevv/ok _ D _) (vfreezevv/ok _ E _) (vvfreezevv/ok _ F _).

% since we'll want to do translation on typing derivations, we'll
% need to know that we can freeze anything.

freeze-gimme     : {N : ccval -> ccexp}   {Z : freeze   N N' F}   type.
vfreeze-gimme    : {N : ccval -> ccval}   {Z : vfreeze  N N' F}   type.
vvfreeze-gimme   : {N : ccval -> ccvval}  {Z : vvfreeze N N' F}   type.
freezevv-gimme   : {N : ccvval -> ccexp}  {Z : freezevv   N N' F} type.
vfreezevv-gimme  : {N : ccvval -> ccval}  {Z : vfreezevv  N N' F} type.
vvfreezevv-gimme : {N : ccvval -> ccvval} {Z : vvfreezevv N N' F} type.

%mode freeze-gimme +N -D.
%mode vfreeze-gimme +N -D.
%mode vvfreeze-gimme +N -D.
%mode freezevv-gimme +N -D.
%mode vfreezevv-gimme +N -D.
%mode vvfreezevv-gimme +N -D.

- : freeze-gimme ([v] cchalt) fz/halt.
- : freeze-gimme ([v] ccgo _ (V1 v) (V2 v)) (fz/go Z2 Z1)
 <- vfreeze-gimme V1 Z1
 <- vfreeze-gimme V2 Z2.
- : freeze-gimme ([v] cccall (V1 v) (V2 v)) (fz/call Z2 Z1)
 <- vfreeze-gimme V1 Z1
 <- vfreeze-gimme V2 Z2.
```

```
- : freeze-gimme ([v] ccfst (V v) ([x] N v x)) (fz/fst ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).
- : freeze-gimme ([v] ccsnd (V v) ([x] N v x)) (fz/snd ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).
- : freeze-gimme ([v] ccwapp (V v) _ ([x] N v x)) (fz/wapp ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).
- : freeze-gimme ([v] ccunpack (V v) ([w][x] N v w x)) (fz/unpack ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x}{w} freeze-gimme ([v] N v w x) (ZN w x)).
- : freeze-gimme ([v] ccleta (V v) ([x] N v x)) (fz/leta ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).
- : freeze-gimme ([v] ccput (V v) ([x] N v x)) (fz/put ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).
- : freeze-gimme ([v] cclets (V v) ([x] N v x)) (fz/lets ZN ZV)
 <- vfreeze-gimme V ZV
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).
- : freeze-gimme ([v] cclocalhost ([x] N v x)) (fz/localhost ZN)
 <- ({x} freeze-gimme ([v] N v x) (ZN x)).

- : vfreeze-gimme ([v] cc1) fz/1.
- : vfreeze-gimme ([v] V) fz/closed.
- : vfreeze-gimme ([v] v) fz/var.
- : vfreeze-gimme ([v] ccsh (VV v)) (fz/ch ZZ)
 <- vvfreeze-gimme VV ZZ.
- : vfreeze-gimme ([v] ccvalid (VV v)) (fz/valid ZZ)
 <- vvfreeze-gimme VV ZZ.
- : vfreeze-gimme ([v] ccpack W (V v)) (fz/pack Z)
 <- vfreeze-gimme V Z.
- : vfreeze-gimme ([v] ccwlam ([w] V v w)) (fz/wlam Z)
 <- ({w} vfreeze-gimme ([v] V v w) (Z w)).
- : vfreeze-gimme ([v] ccheld (V v)) (fz/held Z)
 <- vfreeze-gimme V Z.
- : vfreeze-gimme ([v] ccpair (V1 v) (V2 v)) (fz/pair Z2 Z1)
 <- vfreeze-gimme V1 Z1
 <- vfreeze-gimme V2 Z2.
- : vfreeze-gimme ([v] ccclosure ([a][e] N v a e) (V v)) (fz/closure ZV ZN)
 <- vfreeze-gimme V ZV
 <- ({a}{e} freeze-gimme ([v] N v a e) (ZN a e)).

- : vvfreeze-gimme ([v] ccvv ([w] V w v)) (fz/vv Z)
 <- ({w} vfreeze-gimme ([v] V w v) (Z w)).
- : vvfreeze-gimme ([v] VV) fz/vclosed.


- : freezevv-gimme ([v] cchalt) fzvv/halt.
- : freezevv-gimme ([v] ccgo _ (V1 v) (V2 v)) (fzvv/go Z2 Z1)
 <- vfreezevv-gimme V1 Z1
 <- vfreezevv-gimme V2 Z2.
- : freezevv-gimme ([v] cccall (V1 v) (V2 v)) (fzvv/call Z2 Z1)
 <- vfreezevv-gimme V1 Z1
 <- vfreezevv-gimme V2 Z2.
- : freezevv-gimme ([v] ccfst (V v) ([x] N v x)) (fzvv/fst ZN ZV)
 <- vfreezevv-gimme V ZV
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).
- : freezevv-gimme ([v] ccsnd (V v) ([x] N v x)) (fzvv/snd ZN ZV)
 <- vfreezevv-gimme V ZV
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).
- : freezevv-gimme ([v] ccwapp (V v) _ ([x] N v x)) (fzvv/wapp ZN ZV)
 <- vfreezevv-gimme V ZV
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).
- : freezevv-gimme ([v] ccunpack (V v) ([w][x] N v w x)) (fzvv/unpack ZN ZV)
```

```
 <- vfreezevv-gimme V ZV
 <- ({x}{w} freezevv-gimme ([v] N v w x) (ZN w x)).
- : freezevv-gimme ([v] ccleta (V v) ([x] N v x)) (fzvv/leta ZN ZV)
 <- vfreezevv-gimme V ZV
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).
- : freezevv-gimme ([v] ccput (V v) ([x] N v x)) (fzvv/put ZN ZV)
 <- vfreezevv-gimme V ZV
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).
- : freezevv-gimme ([v] cclets (V v) ([x] N v x)) (fzvv/lets ZN ZV)
 <- vfreezevv-gimme V ZV
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).
- : freezevv-gimme ([v] cclocalhost ([x] N v x)) (fzvv/localhost ZN)
 <- ({x} freezevv-gimme ([v] N v x) (ZN x)).

- : vfreezevv-gimme ([v] cc1) fzvv/1.
- : vfreezevv-gimme ([v] V) fzvv/closed.
- : vfreezevv-gimme ([v] ccsh (VV v)) (fzvv/ch ZZ)
 <- vvfreezevv-gimme VV ZZ.
- : vfreezevv-gimme ([v] ccvalid (VV v)) (fzvv/valid ZZ)
 <- vvfreezevv-gimme VV ZZ.
- : vfreezevv-gimme ([v] ccpack W (V v)) (fzvv/pack Z)
 <- vfreezevv-gimme V Z.
- : vfreezevv-gimme ([v] ccwlam ([w] V v w)) (fzvv/wlam Z)
 <- ({w} vfreezevv-gimme ([v] V v w) (Z w)).
- : vfreezevv-gimme ([v] ccheld (V v)) (fzvv/held Z)
 <- vfreezevv-gimme V Z.
- : vfreezevv-gimme ([v] ccpair (V1 v) (V2 v)) (fzvv/pair Z2 Z1)
 <- vfreezevv-gimme V1 Z1
 <- vfreezevv-gimme V2 Z2.
- : vfreezevv-gimme ([v] ccclosure ([a][e] N v a e) (V v)) (fzvv/closure ZV ZN)
 <- vfreezevv-gimme V ZV
 <- ({a}{e} freezevv-gimme ([v] N v a e) (ZN a e)).

- : vvfreezevv-gimme ([v] ccvv ([w] V w v)) (fzvv/vv Z)
 <- ({w} vfreezevv-gimme ([v] V w v) (Z w)).
- : vvfreezevv-gimme ([v] VV) fzvv/closedvv.
- : vvfreezevv-gimme ([v] v) fzvv/var.

%worlds (blockw | blockccv | blockccvv) (freeze-gimme D _) (vfreeze-gimme D _)
   (vvfreeze-gimme D _) (freezevv-gimme D _) (vfreezevv-gimme D _) (vvfreezevv-gimme D _).

%total (D E F) (freeze-gimme D _) (vfreeze-gimme E _) (vvfreeze-gimme F _).
%total (D E F) (freezevv-gimme D _) (vfreezevv-gimme E _) (vvfreezevv-gimme F _).

% now we can do the translation on typing derivations.

cc : {D  : cof  M  W} {D' : ccof M' W} type.
%mode cc +D -D'.

ccv : {D  : cofv  V  A W} {D' : ccofv V' A W} type.
%mode ccv +D -D'.

ccvv : {D  : cofvv  VV  A} {D' : ccofvv VV' A} type.
%mode ccvv +D -D'.

- : ccvv (covv VF) (ccovv VF') <- ({w} ccv (VF w) (VF' w)).
- : ccv (cov_lam [x][wx] WM x wx) (ccov_closure ccov_unit
                                    ([x][xof][e][eof] WM'' x xof)
                                    ([x] f/closed)
                                    ([x] FwrtBOD))
 <- ({x}{wx}{x'}{wx' : ccofv x' A W}{thm : ccv wx wx'}
      cc (WM x wx) (WM' x' wx' : ccof (M' x') W))
 <- freeze-gimme M' (Z : freeze _ _ FwrtBOD)
 <- freeze/ok WM' Z WM''.

- : ccv cov_unit ccov_unit.
```

```
- : ccv (cov_ch WVV) (ccov_ch WVV')
 <- ccvv WVV WVV'.
- : ccv (cov_valid WVV) (ccov_valid WVV')
 <- ccvv WVV WVV'.
- : ccv (cov_pack A WV) (ccov_pack A WV')
 <- ccv WV WV'.
- : ccv (cov_held WV) (ccov_held WV')
 <- ccv WV WV'.
- : ccv (cov_pair WV1 WV2) (ccov_pair WV1' WV2')
 <- ccv WV1 WV1'
 <- ccv WV2 WV2'.
- : ccv (cov_wlam WV) (ccov_wlam WV')
 <- ({w} ccv (WV w) (WV' w)).
- : ccv cov_const ccov_const.

- : cc (co_call D1 D2) (cco_call D1' D2')
 <- ccv D1 D1'
 <- ccv D2 D2'.

- : cc (co_fst WV WN) (cco_fst WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofv x' A W}{thm : ccv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freeze-gimme M' (Z : freeze _ _ F)
 <- freeze/ok WN' Z WN''.

- : cc (co_snd WV WN) (cco_snd WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofv x' A W}{thm : ccv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freeze-gimme M' (Z : freeze _ _ F)
 <- freeze/ok WN' Z WN''.

- : cc (co_leta WV WN) (cco_leta WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofv x' A W}{thm : ccv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freeze-gimme M' (Z : freeze _ _ F)
 <- freeze/ok WN' Z WN''.

- : cc (co_wapp (WV : cofv V (call Af) W) ([x][wx:cofv x _ W] WN x wx))
       (cco_wapp WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofv x' _ W}{thm : ccv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freeze-gimme M' (Z : freeze _ _ F)
 <- freeze/ok WN' Z WN''.

- : cc (co_localhost ([x][wx : cofv x (caddr W) W] WN x wx)) (cco_localhost WN'' F)
 <- ({x}{wx}{x'}{wx' : ccofv x' (caddr W) W}{thm : ccv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freeze-gimme M' (Z : freeze _ _ F)
 <- freeze/ok WN' Z WN''.

- : cc (co_unpack WV WN) (cco_unpack WV' WN'' F)
 <- ccv WV WV'
 <- ({w}{x}{wx}{x'}{wx' : ccofv x' _ _}{thm : ccv wx wx'}
       cc (WN w x wx) (WN' w x' wx'))
 <- ({w} freeze-gimme (M' w) (Z w : freeze _ _ (F w)))
 <- ({w} freeze/ok (WN' w) (Z w) (WN'' w)).

- : cc (co_lets WV WN) (cco_lets WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofvv x' _}{uthm : ccvv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freezevv-gimme M' (Z : freezevv _ _ F)
```

```
 <- freezevv/ok WN' Z WN''.

- : cc (co_put MO WV WN) (cco_put MO WV' WN'' F)
 <- ccv WV WV'
 <- ({x}{wx}{x'}{wx' : ccofvv x' _}{uthm : ccvv wx wx'}
       cc (WN x wx) (WN' x' wx'))
 <- freezevv-gimme M' (Z : freezevv _ _ F)
 <- freezevv/ok WN' Z WN''.

- : cc co_halt cco_halt.

% would like to do this, but would need a fancier metric
% because the lam in the inductive call is not a subderivation.
% - : cc (co_go WA WC) (cco_go WA' WC')
%  <- ccv WA WA'
%  <- ccv (cov_lam ([x][wx : cofv x cunit W] WC)) WC'.

% so instead, build in the lam case:
- : cc (co_go WA WC) (cco_go WA' (ccov_closure ccov_unit
                                    ([x][xof][e][eof] WC')
                                    ([x] f/closed)
                                    ([x] f/closed)))
 <- ccv WA WA'
 <- cc WC (WC' : ccof M' W).

%block blockcc : some {W:world} {A:ctyp}
                 block {x:cval}{xok:cofv x A W}
                       {x':ccval}{x'ok:ccofv x' A W}{thm : ccv xok x'ok}.
%block blockccvv : some {Af:world -> ctyp}
                   block {x:cvval}{xok:cofvv x Af}
                         {x':ccvval}{x'ok:ccofvv x' Af}{thm : ccvv xok x'ok}.

%worlds (blockw | blockcc | blockccvv) (cc _ _) (ccv _ _) (ccvv _ _).
%total (D E F)                         (cc D _) (ccv E _) (ccvv F _).
```

Bibliography

[1] Martín Abadi. Logic in access control. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 228, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1884-2. 7.2.1

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993. ISSN 0164-0925. 7.2.1

[3] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, pages 74–88, 1999. URL `citeseer.ist.psu.edu/article/abadi98secure.html`. 7.2.2

[4] Andreas Abel. A third-order representation of the $\lambda\mu$-Calculus. In S.J. Ambler, R.L. Crole, and A. Momigliano, editors, *Electronic Notes in Theoretical Computer Science*, volume 58. Elsevier, 2001. 2

[5] Andrew Appel. *Compiling With Continuations*. Cambridge University Press, Cambridge, 1991. ISBN 0521416957. 4.6

[6] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993. 5.5.4

[7] John Billings, Peter Sewell, Mark Shinwell, and Rok Strnisa. The implementation of hashcaml, 2006. URL `http://www.cl.cam.ac.uk/~pes20/hashcaml/`. 7.1.2

[8] John Billings, Peter Sewell, Mark Shinwell, and Rok Strnisa. Type-safe distributed programming for OCaml. In *ML Workshop 2006*, September 2006. URL `http://www.cl.cam.ac.uk/~pes20/hashcaml/`. 7.1.2

[9] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, 1997. URL `http://citeseer.nj.nec.com/blumofe97adaptive.html`. 2.1

[10] Tijn Borghuis and Loe M. G. Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4):274–289, 2000. 7.1.1

[11] Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. Cascading style sheets level 2 revision 1 (css 2.1) specification. Technical Report CR-CSS21-20070719,

W3C, July 2007. URL http://www.w3.org/TR/2007/CR-CSS21-20070719. 6.1

[12] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. In *ICDT '92: Selected papers of the fourth international conference on Database theory*, pages 3–48, Amsterdam, The Netherlands, 1995. Elsevier Science Publishers B. V. 7.2.3

[13] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). In *Theoretical Aspects of Computer Software (TACS)*, pages 1–37. Springer-Verlag LNCS 2215, October 2001. 7.1.1

[14] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part II). In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR)*, pages 209–225, Brno, Czech Republic, August 2002. Springer-Verlag LNCS 2421. 7.1.1

[15] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL)*, pages 365–377. ACM Press, 2000. 7.1.1

[16] B. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless grid computing in ConCert. In M. Parashar, editor, *Grid Computing – Grid 2002 Third International Workshop*, pages 112–125. Springer-Verlag, November 2002. URL http://tom7.org/papers/. 2.1

[17] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon, April; revised December 2003. URL http://www.cs.cmu.edu/~fp/papers/CMU-CS-03-131R.pdf. 3.5.1

[18] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10): 519–538, 2005. ISSN 0362-1340. 7.2.1

[19] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007. URL http://www.sosp2007.org/papers/sosp173-myers.pdf. 7.1.3

[20] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending java for high-level web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003. ISSN 0164-0925. 7.2.3

[21] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer-Verlag, November 2006. To appear. 4.5, 7.1.3, 7.2.3

[22] R. Cooper and C. Krumvieda. Distributed programming with asynchronous ordered channels in distributed ML. In *Proceedings of the ACM SIGPLAN workshop*

on ML and its Applications, June 1992. 7.1.2

[23] Karl Crary. Toward a foundational typed assembly language (expanded version). Technical Report CMU-CS-02-196, Carnegie Mellon University, 2002. 7.2.2

[24] Karl Crary. Linear logic. *The Twelf Wiki*, 2007. URL `http://twelf.plparty.org/wiki/Linear_logic`. 4.7.1

[25] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. *ACM Transactions on Computational Logic*, 2007. To appear. 7.2.2

[26] Curry. The functional logic language Curry, 2007. URL: `http://www.informatik.uni-kiel.de/~curry`. 7.2.3

[27] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982. 5.3.3

[28] R. Davies. A temporal-logic approach to binding-time analysis. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 184, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7463-6. 7.2.1

[29] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001. ISSN 0004-5411. 7.2.1

[30] Maarten de Rijke and Heinrich Wansing. Proofs and expressiveness in alethic modal logic. In *A Companion to Philosophical Logic*, pages 422–441. 3.1.2

[31] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon, May 2005. Available as CMU Technical Report CMU–CS–05–131. 7.2.2

[32] ECMA. ECMAScript language specification. Technical Report ECMA-262, ECMA, December 1999. 5.1.1, 5.4.9

[33] Sameh El-Ansary, Donatien Grolaux, Peter Van Roy, and Mahmoud Rafea. Overcoming the multiplicity of languages and technologies for web-based development using a multi-paradigm approach. In *Multiparadigm Programming in Mozart/Oz, Second International Conference (MOZ 2004), Revised Selected and Invited Papers*, pages 113–124, October 2004. 7.1.3

[34] M. Elsman and K. F. Larsen. Typing XHTML web applications in ML. In *Practical Aspects of Declarative Languages PADL*, pages 224–238, 2004. 7.2.3

[35] E. Allen Emerson. *Temporal and modal logic*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-444-88074-7. 7.2.1

[36] A. P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958. ISSN 0001-0782. 5.5.4

[37] Jim Farley. *Java Distributed Computing*. O'Reilly, January 1998. ISBN 1595922069. 2.2

[38] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Updated by RFC

2817. 5.5.1

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2817 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2817.txt`. 5.5.1

[40] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996. URL `http://citeseer.csail.mit.edu/fournet95reflexive.html`. 7.1.2

[41] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies. RFC 2045 (Draft Standard), November 1996. URL `http://tools.ietf.org/html/rfc2045`. 5.5.4

[42] Richard Frost and John Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal, Special edition on Lazy Functional Programming*, 32:108–121, 1989. 5.3.1

[43] Philippa Gardner, Gareth Smith, Mark Wheelhouse, and Uri Zarfaty. DOM: Towards a formal specification. In *PLAN-X 2008: Programming Language Techniques for XML*, January 2008. URL `http://gemo.futurs.inria.fr/events/PLANX2008/papers/p18.pdf`. 7.2.3

[44] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th Computer Security Foundations Workshop (CSFW'06)*, pages 183–293, July 2006. 7.2.1

[45] Jesse James Garrett. Ajax: A new approach to web applications, February 2005. URL `http://adaptivepath.com/ideas/essays/archives/000385.php`. 7.2.3

[46] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987. 4.7.1

[47] Timothy G. Griffin. The formulae-as-types notion of control. In *Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan 1990*, pages 47–57. ACM Press, New York, 1990. URL `http://citeseer.ist.psu.edu/griffin90formulaeastypes.html`. 3.4.1

[48] Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 27–38, New York, NY, USA, 2006. ACM. ISBN 1-59593-388-3. 7.2.3

[49] Robert Harper. *Practical Foundations for Programming Languages*. Working draft, 2007. URL `http://www.cs.cmu.edu/~rwh/plbook/book.pdf`. 5.4.1

[50] Robert Harper and Karl Crary. How to believe a Twelf proof, 2006. URL `http://www.cs.cmu.edu/~rwh/papers/how/believe-twelf.pdf`. 4.4.1

[51] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January

1993. 4.4.1

[52] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 2007. URL `http://www.cs.cmu.edu/~rwh/papers/mech/jfp07.pdf`. To appear. 4.4.1, 4.4.1, 4.4.1

[53] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL)*, pages 130–141, San Francisco, California, January 1995. 7.1.2

[54] Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–346. MIT Press, 2005. 5.1.3

[55] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000. 7.2.2

[56] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDPi: A language for controlling mobile code. Report 02/2003, Department of Computer Science, University of Sussex, October 2003. 7.1

[57] Michael Huth and Mark Ryan. *Logic in computer science*. Cambridge University Press, 2004. ISBN 0-521-54310X. URL `http://www.cs.bham.ac.uk/research/projects/lics/`. 7.2.1

[58] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, 1992. 5.3.1

[59] hylo.loria.fr. Hybrid logics bibliography, 2005. URL: `http://hylo.loria.fr/content/papers.php`. 3.1

[60] W3C DOM IG. Document object model, January 2005. URL: `http://w3c.org/DOM/`. 5.1.3

[61] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th ACM World Wide Web Conference (WWW 2006)*, 2006. URL `http://crypto.stanford.edu/sameorigin/sameorigin.pdf`. 5.5.2

[62] Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). *European Symposium on Programming*, 2004. URL `http://www.cs.princeton.edu/sip/pub/modal-esop04.pdf`. 3.1, 3.2.1, 3.5.4, 7.1.1

[63] JoCaml. JoCaml web site, 2005. URL: `http://moscova.inria.fr/jocaml/`. 7.1.2

[64] Clifford Dale Krumvieda. *Distributed ML: abstractions for efficient and fault-tolerant programming*. PhD thesis, Cornell University, Department of Computer Science, Ithaca, N.Y., August 1993. URL `http://hdl.handle.net/1813/6150`. COR 93-1376. 7.1.2

[65] Avijit Kumar and Robert Harper. A language for access control. Technical Report CMU–CS–07–140, Carnegie Mellon University School of Computer Science,

Pittsburgh, PA, July 2007. 7.2.1

[66] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *POPL '07: Proceedings of the 34$^{\text{th}}$annual ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, pages 173–184, New York, NY, USA, January 2007. ACM Press. ISBN 1-59593-575-4. URL `http://www.cs.cmu.edu/~dklee/papers/tslf-popl.pdf`. 4.7.1

[67] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. Technical Report CMU-CS-06-138, Carnegie Mellon, January 2007. URL `http://www.cs.cmu.edu/~dklee/papers/tslf.pdf`. 4.7.1

[68] Daniel K. Lee and *et al*. Effectiveness lemma. *The Twelf Wiki*, 2007. URL `http://twelf.plparty.org/wiki/Effectiveness_lemma`. 4.4.2

[69] Håkon Wium Lie and Bert Bos. Cascading style sheets, level 1. Technical Report REC-CSS1-19990111, W3C, Jan 1999. URL `http://www.w3.org/TR/1999/REC-CSS1-19990111`. 6.1

[70] V. Wiktor Marek, Grigori Schwarz, and Miroslaw Truszczynski. Ranges of strong modal nonmonotonic logics. In *Proceedings of the 1st International Workshop on Nonmonotonic and Inductive Logic*, pages 85–99, London, UK, 1991. Springer-Verlag. ISBN 3-540-54564-6. 3.1.2

[71] MediaWiki. Mediawiki, 2007. URL: `http://mediawiki.org/`. 6.3

[72] Sun Microsystems. Java object serialization specification, 2005. URL `http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html`. 2.2

[73] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353. 7.1.2

[74] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, pages 1–40, 41–71, September 1992. 7.1

[75] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 5.3.3

[76] Robin Milner. *Communication and Concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995. ISBN 0-13-115007-3. 7.1

[77] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997. ISBN 0-262-63181-4. 3.5, 5.1

[78] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceedings of the 23th Symposium on Principles of Programming Languages (POPL)*, pages 271–283, 1996. 4.7

[79] MLton. MLton: Unresolved bugs, 2007. URL: `http://mlton.org/UnresolvedBugs`. 5.3.1

[80] Jonathan Moody. Modal logic as a basis for distributed computation. Technical

Report CMU-CS-03-194, Carnegie Mellon University, October 2003. 7.1.1

[81] Jonathan Moody. Logical mobility and locality types. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation (LOPSTR)*, LNCS. Springer, 2005. URL `http://www.cs.cmu.edu/~jwmoody/doc/pub/2004-LOPSTR-mobility-locality.ps`. 7.1.1

[82] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999. 7.2.2

[83] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21 (3):527–568, May 1999. 4.7

[84] Tom Murphy, VII. The wizard of TILT: Efficient[?], convenient and abstract type representations. Technical Report CMU-CS-02-120, Carnegie Mellon School of Computer Science, 2002. URL `http://reports-archive.adm.cs.cmu.edu/anon/2002/abstracts/02-120.html`. 5.1.4, 5.4.1

[85] Tom Murphy, VII. Grid ML programming with ConCert. In *ML Workshop 2006*, September 2006. URL `http://tom7.org/papers/`. 2.1, 5.2.1, 5.5.1, 5.5.4, 7.2.2

[86] Tom Murphy, VII. Modal types for mobile code (thesis proposal). Technical Report CMU-CS-06-112, Carnegie Mellon, Pittsburgh, Pennsylvania, USA, Feb 2006. URL `http://tom7.org/proposal/`. 3

[87] Tom Murphy, VII, Karl Crary, and Robert Harper. Distributed control flow with classical modal logic. In Luke Ong, editor, *14th Annual Conference of the European Association for Computer Science Logic (CSL 2005)*, Lecture Notes in Computer Science. Springer, August 2005. URL `http://tom7.org/papers/`.

[88] Tom Murphy, VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, November 2007. URL `http://tom7.org/papers/`. 5

[89] Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Press, July 2004. URL `http://tom7.org/papers/`. 3.3

[90] Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing (extended technical report). Technical Report CMU-CS-04-105, Carnegie Mellon University, Mar 2004. URL `http://tom7.org/papers/`. 3.2.3

[91] Tom Murphy, VII, Daniel Spoonhower, Chris Casinghino, Daniel R. Licata, Karl Crary, and Robert Harper. The cult of the bound variable: The 9$^{\text{th}}$annual ICFP programming contest. Technical Report CMU-CS-06-163, Carnegie Mellon, October 2006. URL `http://reports-archive.adm.cs.cmu.edu/anon/2006/`

abstracts/06-163.html. 5.4.1

[92] Chetan Murthy. Classical proofs as programs: How, what and why. Technical Report TR91-1215, Cornell University, 1991. 3.4.1

[93] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow, July 2001. Software release, URL: http://www.cs.cornell.edu/jif. 7.1.3

[94] Aleksandar Nanevski, 2006. Personal communication and locally circulated document. 3.4.2

[95] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997. URL http://citeseer.ist.psu.edu/50371.html. 7.2.2

[96] O'Caml. O'caml online documentation, 2005. URL: http://caml.inria.fr/. 7.1.2

[97] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in an polymorphic language. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 99–112, Charleston, South Carolina, 1993. URL http://citeseer.ist.psu.edu/ohori93semantics.html. 7.1.2

[98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998. URL http://citeseer.ist.psu.edu/okasaki98purely.html. 5.1.4

[99] Michel Parigot. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference LPAR'92, St. Petersburg, Russia, July 15–20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*. Springer, 1992. ISBN 3-540-55727-X. 3.4.1

[100] Sungwoo Park. A modal language for the safety of mobile values. Technical Report CMU-CS-05-124, Carnegie Mellon, Pittsburgh, Pennsylvania, USA, May 2005. 3.5.4, 7.1.1

[101] Sungwoo Park. A modal language for the safety of mobile values. In *Fourth ASIAN Symposium on Programming Languages and Systems*, November 2006. 3.5.4, 7.1.1

[102] Sungwoo Park. Type-safe higher-order channels in ML-like languages. In *International Conference on Functional Programming (ICFP) 2007*, October 2007. URL http://www.postech.ac.kr/~gla/paper/icfp056-park.pdf. 7.1.1

[103] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003. ISBN 9780521826143. 7.2.3

[104] Frank Pfenning. Structural cut elimination: I. Intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000. URL http://citeseer.

ist.psu.edu/pfenning00structural.html. 3.2.3

[105] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001. URL http://www.cs.cmu.edu/~fp/papers/handbook01.pdf. 4.4.1

[106] Frank Pfenning. *Automated Theorem Proving*. Carnegie Mellon University, 2004. URL http://www.cs.cmu.edu/~fp/courses/atp/. Draft. 3.4.3, 4.7

[107] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, July 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99). 7.1.1

[108] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, 1988. URL http://www.cs.cmu.edu/~fp/papers/pldi88.pdf. 4.7.1

[109] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language, and interaction: essays in honour of Robin Milner*, pages 455–494. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-16188-5. 7.1

[110] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Programme Construction. 5th International Conference (MPC2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, July 2000. 5.4.1

[111] David Plainfossé and Marc Shapiro. A survey of distributed collection techniques. Technical report, BROADCAST, 1994. 7.2.2

[112] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975. URL http://homepages.inf.ed.ac.uk/gdp/publications/cbn_cbv_lambda.pdf. 4.6

[113] Twelf Project. %total. *The Twelf Wiki*, 2007. URL http://twelf.plparty.org/wiki/%25total. A.3.1

[114] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995. URL http://citeseer.ist.psu.edu/rao95bdi.html. 7.2.1

[115] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. 7.1.2

[116] Greg Restall. Proofnets for S5: sequents and circuits for modal logic. *Logic Colloquium 2005*, (28), 2007. 3.1.2

[117] Andreas Rossberg. *Typed Open Programming - A higher-order, typed approach to dynamic modularity and distribution*. PhD thesis, Universität des Saarlandes, January 2007. URL http://www.mpi-sws.mpg.de/~rossberg/publications.

html. (preliminary version). 7.1.2

[118] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming, Volume 5*, volume 5 of *Trends in Functional Programming*. Intellect, Munich, Germany, 2004. URL http://www.mpi-sws.mpg.de/~rossberg/publications.html. 7.1.2

[119] Andreas Rossberg, Guido Tack, and Leif Kornstaedt. HOT pickles, and how to serve them. In *ACM-SIGPLAN Workshop on ML*, October 2007. URL http://www.mpi-sws.mpg.de/~rossberg/publications.html. 7.1.2

[120] Jesse Ruderman. The same origin policy, 2001. URL: http://www.mozilla.org/projects/security/components/same-origin.html. 5.5.2, 7.2.1

[121] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758. URL http://www.cs.cmu.edu/~fp/papers/tphols03.pdf. 3.3, 4.4.1

[122] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, October 2006. URL http://saxo.essi.fr/~gallesio/Publis/dls06.pdf. 7.1.3, 7.2.3

[123] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *International Conference on Functional Programming (ICFP) 2005*, Tallinn, Estonia, September 2005. 7.1.2

[124] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4-5):547–612, 2007. ISSN 0956-7968. 7.1.2

[125] Peter Sewell, Pawel Wojciechowski, and Benjamin Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical Report 462, Computer Laboratory, University of Cambridge, April 1999. 7.1

[126] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994. 3.1, 3.2

[127] G. Smolka. The Oz programming model. *Computer Science Today*, 1000:324–343, 1995. 7.1.3

[128] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations, August 1999. URL http://tools.ietf.org/html/rfc2663. 7.2.1

[129] David Swasey, Tom Murphy, VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. In *ML Workshop 2006*, September 2006. URL http://tom7.org/papers/. 5.1, 5.1.3

[130] David Swasey, Tom Murphy, VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML (revised and expanded). Technical Report CMU-CS-06-104R, Carnegie Mellon University, January 2006. URL http://reports-archive.adm.cs.cmu.edu/anon/2006/abstracts/06-104R.html. 5.1, 5.1.3, 5.3.3

[131] Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005. ISSN 1533-5399. 7.2.3

[132] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 278–298, London, UK, 1996. Springer-Verlag. ISBN 3-540-61604-7. 7.1.2

[133] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz Knabe, and Alessandro Giacalone. Facile Antigua release programming guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany, December 1993. 7.1.2

[134] TILT. The TILT compiler, 2004–2005. URL: http://www.tilt.cs.cmu.edu/. 5.3.3

[135] Asis Unypoth and Peter Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, London, UK, January 2001. 7.1

[136] C. Joseph Vanderwaart. *Static Enforcement of Timing Policies Using Code Certification*. PhD thesis, Carnegie Mellon University, August 2006. URL http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-143.pdf. Available as CMU Technical Report CMU–CS–06–143. 5.5.3

[137] Joseph C. Vanderwaart, Derek R. Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. Typed compilation of recursive datatypes. In *TLDI '03: 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 98–108, January 2003. URL http://www.cs.cmu.edu/~rwh/papers/datatypes/tldi.pdf. 5.1.4, 3

[138] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997. URL http://citeseer.ist.psu.edu/vinoski97corba.html. 2.2

[139] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages (POPL)*, pages 307–312. Association for Computing Machinery, 1987. 5.1.4

[140] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the 8th International Conference on Functional Programming (ICFP)*. ACM Press, August 2003. URL http://homepages.inf.ed.ac.uk/wadler/topics/dual.html. 3.4.2

[141] Philip Wadler. Call-by-value is dual to call-by-name, reloaded. In *Rewriting Techniques and Applications*, April 2005. URL `http://homepages.inf.ed.ac.uk/wadler/topics/dual.html`. 3.4.2

[142] Ken Wakita, Takashi Asano, and Masataka Sassa. D'Caml: A native distributed ML compiler for heterogeneous environment. In Patrick Amestoy, Philippe Berger, Michel J. Daydé, Iain S. Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par 1999 Parallel Processing, 5th International Euro-Par Conference*, volume 1685 of *Lecture Notes in Computer Science*, pages 914–924. Springer, 1999. ISBN 3-540-66443-2. URL `http://citeseer.ist.psu.edu/wakita99dcaml.html`. 7.1.2

[143] Wikipedia. Wikipedia, the free encyclopedia, 2007. URL: `http://en.wikipedia.org/`. 6.3

[144] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000. ISSN 0956-7968. 7.2.3

[145] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing (a state-of-the-art report on OMG/CORBA). *Operating Systems Review*, 30(2):4–31, 1996. URL `http://citeseer.ist.psu.edu/yang96corba.html`. 2.2