Translucent Sums: A Foundation for Higher-Order Module Systems

Mark Lillibridge

May, 1997

CMU–CS–97–122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
Robert Harper, Chair
Peter Lee
John Reynolds
Luca Cardelli, DEC SRC

This research was sponsored by the Air Force Materiel Command (AFMC) and the Defense Advanced Research Projects Agency (DARPA) under contract number, F19628-95-C-0050. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

Keywords: type theory, modularity, abstraction, sharing, functional programming, lambda calculus, Standard ML, modules and interfaces, packages, first-class modules, abstract data types.

Abstract

The ease of understanding, maintaining, and developing a large program depends crucially on how it is divided up into modules. The possible ways a program can be divided are constrained by the available modular programming facilities ("module system") of the programming language being used. Experience with the Standard-ML module system has shown the usefulness of functions mapping modules to modules and modules with module subcomponents. For example, functions over modules permit abstract data types (ADTs) to be parameterized by other ADTs, and submodules permit modules to be organized hierarchically. Module systems with such facilities are called higher-order, by analogy with higher-order functions.

Previous higher-order module systems can be classified as either *opaque* or *transparent*. Opaque systems totally obscure information about the identity of type components of modules, often resulting in overly abstract types. This loss of type identities precludes most interesting uses of higher-order features. Transparent systems, on the other hand, completely reveal type identities by inspecting module implementations, which subverts data abstraction and prevents separate compilation.

In this dissertation, I describe a novel approach that avoids these problems based on a new type-theoretic concept, the translucent sum. A translucent sum is a kind of weak sum whose type can optionally specify the identity of its type components. Under my approach type identities are not determined by inspecting module implementations, permitting separate compilation. By default, module operations reveal the connections between the types in their input and output modules. However, these connections can be obscured using type coercion. This controlled visibility permits data abstraction where desired without limiting the uses of higher-order features. My approach also allows modules to be treated as first-class values, a potentially valuable feature.

In order to lay out the groundwork for my new approach to designing higher-order module systems and to demonstrate its utility, I develop in complete detail a kernel system using my approach. I establish formally the theoretical properties of the system, including soundness even in the presence of side effects. I also show how the system may be effectively type checked.

To Donna Jean

Contents

Acknowledgements x

I Context **1**

1 Introduction **2**

1.1 Overview . . . 2
1.2 Module Systems . . . 3
1.3 Benefits of Higher-Order Module Systems . . . 5
1.4 Example: B-Trees . . . 7

2 Previous Approaches **10**

2.1 The Opaque Approach . . . 10
2.2 The Transparent Approach . . . 12
2.3 Summary . . . 14

3 My Approach **16**

3.1 Module and Functor Boundaries . . . 16
3.2 Default Interfaces . . . 17
3.3 Subtyping . . . 18
3.4 The B-Tree Example . . . 18
3.5 The Recipe . . . 20
3.6 Translucent Sums . . . 20
3.7 First-Class Values . . . 21
3.8 Linking . . . 22
3.9 Type-Sharing Specifications . . . 26
3.10 Summary . . . 27

4 Technical Machinery **28**

4.1 Dot Notation . . . 28

4.2 Creation 30
4.3 Equality 32
4.4 Subtyping 34
4.5 The Phase Distinction 37
4.6 Selection and Application 39
4.7 The EVALUE rules 43

II The Kernel System **47**

5 Overview **48**
5.1 Higher Kinds 49
5.2 Previous Work 51
5.3 Extended Interface Matching 52
5.4 Unnamed Binary Sums 52
5.5 Reified Constructors 54
5.6 Rules and Restrictions 56
5.7 Notation 57
5.8 System Summary 57
5.8.1 Module Constructs 59
5.8.2 References and Recursive Types 60
5.8.3 Semantics 61
5.9 On the Nature of the Proofs 61
5.10 Organization 62

6 Types: Setup **65**
6.1 Kinds and Constructors 65
6.2 Proof Strategy 68
6.3 The Type Rules 71
6.4 Place Lookup 74
6.5 Basic Propositions 77
6.6 Tagging 79
6.7 The Tagged System 81
6.8 Basic Tagged Propositions 83

7 Types: Confluence **88**
7.1 Rewriting 88
7.2 The Proof Method 89
7.3 The Rewrite Relation 91
7.4 Argument Kind Erasure 97

7.5 Kind Preservation 102
7.6 Subject Reduction 110
7.7 Confluence 119
7.8 Equality 120

8 Types: Canonicalization **125**
8.1 Normal Forms 125
8.2 The Method 127
8.3 $\beta\eta$-Normalization 129
8.4 Canonical Form 138
8.5 Assignment Reduction 142
8.6 Decidability 144

9 Types: Summary **156**
9.1 Tag Removal 156
9.2 Stamping 160
9.3 System Correspondence 166
9.4 Converted Results 170
9.5 Semi-Canonical Types 174
9.6 Strengthening 178
9.7 Alternatives 180

10 Subtyping **182**
10.1 One-Step Subtyping 183
10.2 The Subtyping Relation 185
10.3 Shapes 186
10.4 Replacement by a Subtype 190
10.5 A Semi-Decision Procedure 196
10.6 The Simple Type System 200
10.7 Encoding Simple Judgments 205
10.8 Problem Reduction 209
10.9 Undecidability 218

11 Soundness **225**
11.1 The Terms 225
11.2 Self Validity 229
11.3 Self and Subtyping 233
11.4 Runtime States 241
11.5 Properties of Values 248
11.6 Value Substitution 252

11.7 Evaluation 258

12 Type Checking **266**
12.1 Type-Checking Difficulties 266
12.2 The Inference System 267
12.3 Semi-Decidability 271

III Conclusions **276**

13 Extensions **277**
13.1 N-Ary Named Fields 277
13.2 Extended Interface Matching 279
13.3 Transparent Value Declarations 282
13.4 Kind Components 286
13.5 Recursion 287
13.6 Better Type Inference 290
13.7 Elaborations 290
13.7.1 SML-like Elaborations 291
13.7.2 Translucent-Sum Syntax 291
13.7.3 Open and Include 292
13.7.4 Leroy's With Notation 293
13.7.5 Separate Compilation 294
13.8 Reformulations 294

14 Related Work **299**
14.1 First-Order Module Systems 299
14.2 Previous Higher-Order Module Systems 300
14.2.1 Modula-3 301
14.2.2 Mesa and Cedar 302
14.2.3 Standard ML 302
14.2.4 Experimental Designs 304
14.3 Manifest Types 305
14.4 Modules as Second-Class Values 305
14.5 Other Follow-On Work 310

15 Conclusions **312**
15.1 Contributions 312
15.1.1 Better Higher-Order Module Systems 312
15.1.2 Feasibility of My Approach 313

15.1.3 Technical Contributions 314
15.2 Future work 315

A Collected Kernel System Rules 316
A.1 The Kind and Constructor Level 316
A.2 The Term Level 322

Bibliography 326

Acknowledgements

I have many people to thank for helping me complete this dissertation. First, and foremost, of course, is my advisor, Robert Harper. I blame his unwavering high standards for the overall quality of this work. Not content to ensure the technical quality, he also encouraged me to write as clearly as possible; his recommendation of Lyn Dupreé's book *Bugs in Writing* was particularly helpful in this regard. I have learned a great deal from Bob over the years and will miss the long technical arguments I had with him.

I also owe a large debt to the other members of my thesis committee: Peter Lee, John Reynolds, and Luca Cardelli. They provided invaluable advice in a timely fashion. I particularly want to thank them for letting me put off writing the introduction until the very end. John Reynolds also deserves special mention for acting as my advisor during my first year at CMU.

In addition to the people I have already mentioned, I have benefited from technical discussions over the years with Mark Leone, Dave MacQueen, Greg Morrisett, Frank Pfenning, Benjamin Pierce, Chris Stone, Doug Tygar, Dave Tarditi, and the rest of the Pop and Fox groups at CMU. I want to thank Dana Scott for his key role in assembling the strong group of programming-language researchers at CMU. I would also like to thank Cleah Schlueter and Sharon Burks for providing non-technical help on more occasions than I can count.

Thanks go to Andrew Appel, Olivier Danvy, Andrew Gordon, Nick Haines, Rustan Leino, Xavier Leroy, Brian Milnes, John Mitchell, and Mads Tofte for providing feedback on drafts of this work. I would like to thank the National Science Foundation (NSF) for funding me for my first three years of graduate school via a fellowship. My new boss, Roy Levin, at DEC SRC also deserves thanks for forcing me to work on finishing my dissertation full time at SRC; without this motivation, I am certain the process of finishing my dissertation would have dragged on far longer due to the abundant distractions available at SRC.

One of the most important influences on the quality of a graduate student's daily life is his or her officemates. Fortunately, I have been blessed with a series of great officemates: Eka Ginting, Mark Leone, Tom Kang, Darryl Kindred, Bill Niehaus, Lu Qi, and John Greiner (honorary). Eka and Tom deserve special mention for their unflagging efforts to distract me from my dissertation, and indeed, work of any kind. In the process, Eka managed to teach me a lot about business, which I hope to put to good use someday.

Working on a Ph.D. dissertation can be a extremely frustrating and stressful experience. My primary stress release while working on my dissertation was to go ballroom or swing dancing. Accordingly, I would like to thank the Carnegie Mellon University Ballroom Dance Club and the Edgewood Club swing community for providing me the opportunity to dance up a storm. My friends Donna Jean Kaiser, Cheryl Pope, Jim Reich, and James Rauen also helped keep me sane during this trying period. Finally, I would like to thank my family for everything they have done for me over the years.

Mark Lillibridge,
April 24, 1997

Part I

Context

Chapter 1

Introduction

1.1 Overview

Many programming languages have a collection of facilities for building modular programs. These collections of facilities, called *module systems*, play an important role in programming languages, especially with regard to programming-in-the-large. This dissertation concerns a new approach to designing module systems for statically-typed programming languages. The approach promises a substantially more powerful module system than those built with previous approaches, while at the same time eliminating the problems associated with the previous approaches.

Traditional module systems, such as the one provided by Modula-2 [56], are *first-order*, allowing only trivial manipulations of modules. Some newer programming languages provide *higher-order* module systems. Higher-order module systems, unlike first-order ones, permit the non-trivial manipulation of modules within the language. In particular, they at least permit functions mapping modules to modules and may provide other higher-order features such as modules containing modules as subcomponents and modules as first-class values. Experience with the Standard-ML module system has shown the usefulness of functions mapping modules to modules and modules with module subcomponents. For example, functions over modules permit abstract data types (ADTs) to be parameterized by other ADTs, and submodules permit modules to be organized hierarchically. Modules as first-class values is also likely to be useful because it permits choosing ADT implementations at runtime.

Previous higher-order module systems can be classified as either *opaque* or *transparent*. Opaque systems totally obscure information about the identity of type components of modules, often resulting in overly abstract types. This loss of type identities precludes most interesting uses of higher-order features. Transparent systems, on the other hand, completely reveal type identities by inspecting module implementations, which subverts data abstraction and prevents separate compilation. Unlike opaque systems, transparent systems cannot support modules as first-class values. My thesis is as follows:

> By basing a module system on a new type-theoretic concept, namely a new kind of weak sum whose type can contain both transparent and opaque type declarations called a *translucent sum*, it is possible to obtain a higher-order module system with the advantages of both the opaque and the transparent approaches, but none of their disadvantages.

In order to demonstrate this thesis, I design a new programming language with a higher-order module system based on translucent sums. The language I have created is a kernel system that contains only the features relevant to module system design; considerable extension would be required to make it into a real programming language. The design of this kernel system and the proofs required to establish formally its properties form the core of this dissertation. Included are a proof of the system's soundness and effective procedures for type checking its programs.

This dissertation is divided into three parts. The first part introduces the problem and provides the needed background on module systems (this chapter), explains the previous approaches to designing higher-order module systems (Chapter 2), and describes my approach using translucent sums (Chapters 3 and 4). The second part, comprising Chapters 5 to 12, is devoted to the kernel system and its associated proofs. An overview of this part can be found in Chapter 5; descriptions of this part's other chapters can be found in Section 5.10. Finally, the third part discusses possible extensions to the kernel system (Chapter 13), explains related work (Chapter 14), and summarizes this dissertation's results (Chapter 15).

1.2 Module Systems

A *module* binds a set of values and types to names. For example, we might have

```
M = module
        val  x = 3;
        val  y = true;
        type T = int;
    end;
```

This code creates a new module with three components, called `x`, `y`, and `T`. The components `x` and `y` are value components that are bound to the values `3` and `true` respectively; the component `T`, by contrast, is a type component that is bound to the type `int`. Once this module is created, it is then bound to the name `M`; this binding will allow us to refer to the new modules' components as `M.x`, `M.y`, and `M.T` later on. Although in this example, we named our new module, this is not required. Sometimes, for example, we may want to create a new module then immediately apply a function to it; naming the new module serves no purpose in that case.

Because I am dealing with statically typed languages, modules will have "types", called *interfaces*. For example, the module `M` matches the following interface:

```
M : interface
        val  x:int;
        val  y:bool;
        type T;
    end;
```

This interface specifies that `M` has two value components, `x` with type `int` and `y` with type `bool`, and one type component, `T`.[1] The relationship between modules and interfaces is many-to-many: Many modules may match the same interface and a given module may match many interfaces.

A programming language's collection of facilities for building modular programs is its module system. I classify module systems into two types, depending on what sort of facilities they have. *First-order* module systems have only the trivial module facilities discussed so far: module creation, module naming, and module-component extraction (`M.x`). Most traditional module systems are of this type. Examples include Ada [52], CLU [36], C [29], C++ [53], and Modula-2 [56].

Higher-order module systems, by contrast, have non-trivial module manipulation facilities. I shall be concerned in this dissertation primarily with three such facilities: *functors*, *submodules*, and modules as first-class values. Functors are functions mapping modules to modules. For example, we might define a rectangle ADT parameterized by a point ADT using a functor:

```
MkRect = functor(p:POINT):RECT
            module
                type T = p.T * p.T;
                ...
             end;
```

The advantage of doing this is that we can create several rectangle ADTs from the same code using different point ADTs. For example, if we have point ADTs `CartPoint` and `PolarPoint`, we can create a rectangle ADT based on Cartesian points with `CartRect`

[1]In my kernel system I deal with a more complicated version of modules that assigns "types", called *kinds*, to types; I have simplified things here.

`= MkRect(CartPoint)` and a rectangle ADT based on polar points with `PolarRect = MkRect(PolarPoint)`.

Submodules are modules contained as components within other modules; they allow packaging up a series of modules into a single unit. For example, if we had several search-tree modules, we could package them up into a single module:

```
SearchTree = module
                 mod Binary   = BinarySearchTree;
                 mod RedBlack = RedBlackSearchTree;
                 mod BTree    = BTree;
             end;
```

`SearchTree`, in turn, could be included as part of a larger library algorithms module. By organizing modules in this way, a hierarchical namespace can be obtained; such namespaces are easier for programmers to deal with than flat namespaces. Because most module systems do not consider modules to be ordinary values, a different keyword (**mod**) is needed in general here to specify that these are module components, not value components.

Modules as first-class values refers to the ability to treat modules as if they were ordinary values of the programming language. This means that anything that can be done with a ordinary value can be done with a module. Thus, in languages with modules as first-class values, modules can be passed to or returned by functions, stored in variables, or selected using conditional statements. In such languages, the **mod** keyword is just syntactic sugar for the **val** keyword.

One useful way to use this facility is to select the most efficient implementation for an ADT at runtime. For example, the best implementation for a dictionary depends on the number of items it will contain; the following code sets `Dictionary` at runtime to use an implementation that is efficient for `n` items:

```
Dictionary = if n<20 then LinkedList else HashTable;
```

1.3 Benefits of Higher-Order Module Systems

Higher-order module systems offer many benefits over first-order module systems. I have already mentioned some benefits (the ability to parameterize with respect to ADTs, the ability to have a hierarchical namespace, and the ability to choose ADT implementations at runtime), but the greatest benefit of higher-order module systems is that they allow programs to be organized better. Program organization is crucial to ensuring that large programs are easy to develop, understand, and maintain.

The most well-known programming language with a higher-order module system is Standard ML (SML) [21]. SML provides functors and submodules, but treats modules as

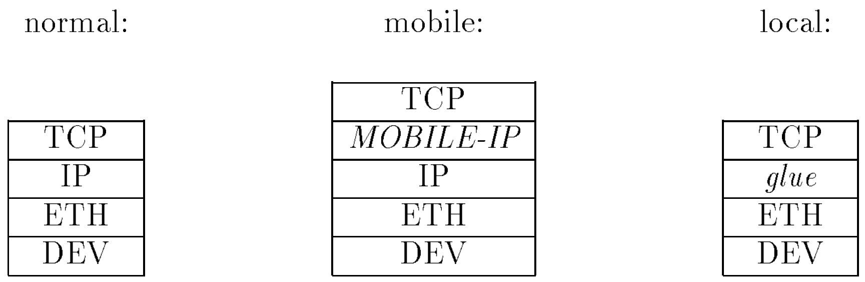

Figure 1.1: TCP/IP protocol organization

second-class values. The extensive experience of the SML community with functors and submodules has established their value as program-organization tools. Modules as first-class values shows promise as a valuable addition to module system toolkits, but since this facility has never been implemented together with the other higher-order facilities, experience about its value is lacking.

It is hard to find good examples of why better organization matters: Any small section of a program can usually be rewritten so that it no longer uses higher-order module system features at the cost of making the rest of the program more complicated. Nonetheless, I am now going to present two examples that I feel convey some of the flavor of how higher-order modules systems are used in practice.

My first example comes from Carnegie Mellon's Fox Project's implementation of the TCP/IP protocol suite, the FoxNet [6, 5, 4], in SML. The TCP/IP protocol suite is organized modularly in the form of a protocol stack (see Figure 1.1). The leftmost stack in this figure shows the organization of normal TCP/IP. Normal TCP/IP is built on a base device driver protocol (DEV), upon which additional protocol layers (ETH, IP, and TCP in turn) are placed. Each of the additional protocol layers takes in the lower level protocol implemented by the underlying stack and produces a new protocol with additional functionality. For example, the IP layer adds the ability to route between different nets.

What is particularly nice about this organization is that the protocol interfaces of the layers are (mostly) identical, allowing protocol layers to be mixed and matched in order to generate many kinds of functionality. For example, the middle stack in Figure 1.1 represents a configuration for mobile TCP/IP, where in an extra protocol layer, MOBILE-IP, has been added between the TCP and IP layers in order to provide an extra layer of indirection between virtual and physical IP addresses; mobile TCP/IP allows computers to move around physically without losing connections.

Alternatively, the rightmost stack in the figure represents a configuration for local TCP/IP, where the IP layer has been dropped. Because the resulting protocol lacks any inter-net routing ability, it can be used only between hosts on the same local net; however, it will be faster for this purpose because it does less processing and uses smaller packets. (A small glue layer, labeled *glue*, is necessary between the TCP and ETH layers because the protocol interfaces these layers use are slightly different in practice.)

By using functors, FoxNet is able to directly capture this modular organization. Idealized code for doing this might look like the following:

```
DEV : PROTOCOL = module ... end;

ETH = functor (lower:PROTOCOL):PROTOCOL
        module ... end;
...
```

Here, the base device layer `DEV` is implemented as a normal module matching the PROTOCOL interface and the other layers are implemented as functors mapping a PROTOCOL to a PROTOCOL. Using this code, the normal and local TCP/IP protocol stacks could then be built up by applying the reverent layer functors in the correct order:

```
Normal = TCP(IP(ETH(DEV)));
Local  = TCP(Glue(ETH(DEV)));
```

In a language with modules as first-class values, we could choose which of these protocol stacks to use at runtime for maximum efficiency:

```
P = if on_local_net() then Local else Normal;
```

Note that the resulting implementation is completely type safe and does not require any runtime type checks. A similar-looking object-oriented programming (OOP) program could be written where protocols and packets are objects. (Each protocol contains a packet type, which is the type of packets belonging to that protocol.). The resulting program would be much slower though because it would require frequent runtime type checks. (Higher-level–protocol packets have extra fields not present in lower-level–protocol packets; runtime type checks must be done before those fields can be accessed because first-order module systems cannot statically distinguish between the different packet types.)

1.4 Example: B-Trees

My second example involves an ordered dictionary implementation. I have chosen this example carefully and will be using it throughout the first part of my dissertation because it illustrates well the differences between the various approaches to building higher-order module systems.

An ordered dictionary is a dictionary whose entries are ordered by an ordering on its keys. The classic example of an ordered dictionary is the white pages telephone directory: You can look up phone numbers by name as well as examine entries in alphabetical order.

It order to define the notion of an ordered dictionary, it is first necessary to define the notion of an *ordered type*:

```
ORDERED = interface type T;
                    val   cmp: T*T -> int;
          end;
```

Here, I have defined an ordered type to be a module containing a type `T`[2], the type in question, together with a comparison function `cmp` that compares two values of type `T`, returning an integer that indicates how they compare. (That is, `cmp(x,y)<0` if `x < y`, `cmp(x,y)=0` if `x = y`, and `cmp(x,y)>0` if `x > y`.)

Using this definition, the notion of an ordered dictionary ADT can then be defined:

```
ORDERED_DICTIONARY =
  interface
     mod Key:ORDERED;
     type elem;
     type T;

     val new:    unit -> T;
     val define: T * Key.T * elem -> unit;
     val first:  T -> Key.T * elem;
     ...
  end;
```

Here, `Key` is the ordered type of keys, `elem` is the type of the elements looked up, and `T` is the type of ordered dictionaries managed by the ADT. The functions `new`, `define`, `first`, and so on, are the operations on ordered dictionaries. For example, `new()` creates a new empty ordered dictionary (the type `unit` contains only one value, denoted `()`), `define(d,k,e)` adds the binding of key `k` to element `e` to the ordered dictionary `d`, and `first(d)` either returns the first binding in the dictionary or raises the exception `fail`.

An example implementation of an ordered dictionary ADT might then be obtained using B-Trees as follows:

[2]By convention, the primary type of a module is called `T`.

```
MkBTree =
  functor (K:ORDERED, E:interface type T; end)
              :ORDERED_DICTIONARY
    module
       mod Key = K;
       type elem = E.T;
       type T = btree(Key.T,elem);

       fun new() = ...
       fun define(dict:T, key:Key.T, value:elem) = ...
       ...
    end;
```

I have parameterized the implementation with respect to which ordered type to use for keys (`K`) and which type to use for elements (`E.T`).[3] The implementation first makes copies of the key and element information so that future clients will have access to this information; second, establishes the representation type it will use, which I have abbreviated here as *btree(*`Key.T`,`elem`*)* so that I can refer to it later easily; and, finally, third, it provides code for each of the required ordered dictionary operations (not shown).

Because the behavior of `MkBTree` varies depending on which approach to designing higher-order module systems is being used, I shall defer further discussion of `MkBTree` until I have described the approaches in more detail.

[3]For technical reasons, it is necessary to wrap up the element type in a module in order to pass it to a functor; this fact is not important here.

Chapter 2

Previous Approaches

In order to motivate my approach to designing higher-order module systems, I shall now describe the previous approaches to designing higher-order module systems. The previous approaches can be classified as either “opaque” or “transparent”, depending on how they treat module and functor boundaries.

2.1 The Opaque Approach

Under the opaque approach, module and functor boundaries are opaque, allowing no information about the identity of type components to pass through. This is exactly what we want when we build an ADT. For example, consider the following definition of an integer stack ADT:

```
IntStack = module
               type T = ref(list(int));
               fun  new() = new [];
               fun  push(s:T, e:int) = ...
               ...
           end;
```

The identity of the integer stack type (`T`) is visible inside the module, allowing us to implement its operations using the appropriate primitives on its representation type, but its identity is obscured outside the module: `IntStack.T` is an abstract type. Thus, the user of this ADT will be able to create and manipulate integer stacks using code like `IntStack.pop(IntStack.push(IntStack.new(), 3))`, which returns `3`, but will not be able to depend in any way on which representation it uses.

Examples of programming languages which take this approach are John Mitchell and Gordon Plotkin’s SOL [43] and Luca Cardelli’s Quest [10]. This approach provides data abstraction at the module level, as we have just seen. It also allows *separate compilation*, the ability to type check and compile individual module implementations using only the interfaces of the modules they reference, because modules depend only on interfaces, not on implementations. This approach is also compatible with modules as first-class values because it makes no assumptions about type-component identities.

Unfortunately, because this approach obscures the identity of *all* type components, it precludes most interesting uses of higher-order features. Consider the B-Tree example I introduced in Section 1.4. Suppose we try to use it to create a telephone directory. To keep things simple, let us use the type `string` as both the key and element type. We can build an ordered type "`string`, ordered alphabetically" by wrapping up `string` with the appropriate `string` comparison operator:

```
StringOrd = module
                type T   = string;
                type cmp = ...;
            end;
```

A simple functor application then suffices to create an ordered dictionary using this ordered type for keys and `string`'s for elements:

```
D = MkBTree(StringOrd, module type T=string; end);
```

As a test of our new ADT, we can try creating a new telephone directory and then adding an entry to it:

```
directory = D.new();

D.define(directory, "John Smith", "123-4567");
```

Unfortunately, the second line above fails to type check under the opaque approach. The reason is that `D.define` has the type

```
D.T * D.Key.T * D.elem ⇔> unit
```

which specifies that `D.define`'s second and third arguments are of type `D.Key.T` and `D.elem` respectively; because the opaque approach treats all type components abstractly outside their module of definition, these two types are not compatible with `string`, the type of the passed second and third arguments. This problem renders the B-Tree implementation unusable under the opaque approach.

The problem with this example is an instance of a more general problem: There is no way in the opaque approach to build modules "containing" other types; we can only build modules containing unrelated new types (e.g., `StringOrd.T` $\neq$ `string`). Hence, modules cannot export type abbreviations; for example, we cannot make `Complex.T` be an abbreviation for the type `Real*Real`. Also, we cannot extend ADTs by adding a few new operations to an existing ADT.[1] Another consequence is that many of the more useful idioms using higher-order features are inexpressible. Indeed, David MacQueen argued in a 1986 paper [37] that this problem is severe enough that the opaque approach should be abandoned in favor of a transparent one.

2.2 The Transparent Approach

Unlike in the opaque approach, under the transparent approach, module and functor boundaries are transparent, allowing all information about the identities of type components to pass through them; the transparent approach does this by inspecting module implementations to determine the actual identity of type components. For example, consider again the `IntStack` example:

```
IntStack = module
               type T = ref(list(int));
               fun  new() = new [];
               fun  push(s:T, e:int) = ...
               ...
           end;
```

This time, the identity of the integer stack type is visible both inside and outside of the module: We know that `IntStack.T = ref(list(int))` and can thus manipulate integer stacks using reference and list primitives. The identity of `IntStack.T` is computed by substituting in its implementation and then reducing the resulting expression:

```
IntStack.T =
(module type T=ref(list(int)); ... end).T =
ref(list(int))
```

Examples of programming languages that take this approach are David MacQueen's DL [37]; Robert Harper and John Mitchell's XML [41]; and Robert Harper, John Mitchell, and Eugenio Moggi's λ^{ML} [22]. As we have just seen, the transparent approach does not provide data abstraction at the module level. It is possible, however, to obtain a more limited kind of data abstraction, which I call *closed* abstraction, under the transparent approach.

[1] For example, we cannot create a new module `ExtendedString` from a built-in `String` ADT module by first copying the `String` type and operations then adding a few new operations, and have extended strings be interchangeable with the built-in strings.

Closed abstraction is abstraction limited to a specific closed scope. For example, we might have an abstract-in construct that allows us to hold our integer-stack module abstract in a particular piece of code (M below):

```
INTSTACK = interface
               type T;
               val  new : unit -> T;
               ...
           end;

abstract IntStack:INTSTACK = module
                                type T = ref(list(int));
                                fun  new() = new [];
                                 ...
                               end
  in
    M
  end;
```

Here `IntStack` is held abstract inside M; that is, only the information available in the interface given for it (`INTSTACK`) is visible inside M. Thus, the fact that `IntStack.T` is a type is visible but not the fact that `IntStack.T` is equal to `ref(list(int))`. Because `IntStack` is only visible in M, this construct provides only closed abstraction.

The abstract-in construct can be desugared in the transparent approach into a functor application:

```
(functor (IntStack:INTSTACK) M)
  (module
     type T = ref(list(int));
     fun  new() = new [];
     ...
   end);
```

Because in general it is not possible to determine the code for functor arguments at compile time, the transparent-system type checker checks M using only the information available in interface for `IntStack`, giving the desired behavior.

By contrast, in *open* abstraction, which is provided by my approach and the opaque approach but not by the transparent approach, the scope of the abstraction is not limited to a single existing scope. For example, using an open-abstraction construct (denoted by **abs** here), we might have the following code:

```
S = module
       abs IntStack:INTSTACK = module
                                 type T = ref(list(int));
                                 fun  new() = new [];
                                 ...
                               end;
       ...
     end
...
```

Here our integer-stack module is held abstract both inside the module `S` (under the name `IntStack`) and outside the module `S` (under the name `S.IntStack`). If `S` is at the top level of a program then the later scope may include modules complied separately from `S` at a later time. Closed abstraction cannot handle such modules because it requires that all code that uses an ADT be available at the time the ADT is written. Because of this important limitation, I do not consider systems that support only closed abstraction to provide abstraction at the module level. Note that under the opaque approach **abs** is just a synonym for **mod** because abstraction is automatic.

Because the transparent approach requires access to the implementations of all the modules a module refers to in order to compile that module (in order to determine the identities of type components from those modules), it cannot provide separate compilation. Also, the transparent approach is incompatible with treating modules as first-values because there is no way to determine the actual identity of a type when it depends on information available only at runtime.[2]

However, the transparent approach does allow modules to "contain" other types. This fact allows most of the interesting use of higher-order features to be used under the transparent approach. For example, if we try and build a telephone directory using `MkBTree` like we did before (see Section 2.1), no type errors occur this time because under the transparent approach, `D.define` has the type *btree(*`string,string`*)* `* string * string -> unit`. Accordingly, the resulting telephone dictionary is fully usable, but its implementation is exposed (note the first argument's type), which is undesirable from the point of view of information hiding.

2.3 Summary

Summarizing, there are two main previous approaches to designing higher-order module systems, the opaque approach and the transparent approach. The opaque approach has

[2]More precisely, unsoundness results from this combination in the presence of side effects; the unsoundness example in Section 4.5 shows this fact because it is well typed under the transparent approach extended with first-class modules but causes a type error at runtime.

the following advantages:

- Provides module-level data abstraction
- Supports separate compilation
- Compatible with modules as first-class values

Its disadvantages include the absence of all of the following:

- Modules can "contain" other types
- Modules may export type abbreviations
- Many of the more useful higher-order idioms work

The transparent approach, on the other hand, has the opposite advantages and disadvantages; it lacks the opaque approach's advantages but it removes the opaque approach's disadvantages. Both approaches' set of disadvantages are serious enough to result in their each building unsatisfactory module systems.

Chapter 3

My Approach

In this chapter I introduce the basic ideas of my approach to designing higher-order module systems, contrast my approach with the previous ones, and explore my approach's abilities. In the next chapter (Chapter 4), I explain in more detail the technical machinery behind my approach.

3.1 Module and Functor Boundaries

Under my approach, which I call the *translucent* approach, module and functor boundaries are "translucent," being opaque in some places and transparent elsewhere. Where the boundaries are opaque is controlled by the programmer using interfaces. Each boundary has an associated interface, which specifies which type component's identity information can be seen through the boundary.

In order to allow this specification, interfaces in my approach are extended to allow specifying the identity of type components. For example, the following interface for a hash table ADT specifies the type identity of the element type (`elem`) but not that of hash tables (`T`):

```
HashTable : interface type elem = int;
                      type T;
                      ...
            end
```

Using this ability, the programmer specifies boundary properties by giving, in the interface associated with a boundary, the type identities of only those components she wishes the boundary to be transparent to; all other type components will be treated opaquely. Thus, in the `HashTable` example above, the boundary will be transparent for the `elem` component and opaque for the `T` component.

Because the programmer is required to provide the actual type identity of transparently-treated type components in the interface (the type checker checks that the type identities provided in the interface are correct when the interface is assigned to a module), the type checker does not need to inspect the implementation of modules to determine the identity of their type components; it can just believe the module's interface. This fact means that modules depend on only the interfaces of other modules; my approach is thus able to support separate compilation without any trouble.

3.2 Default Interfaces

If the programmer does not supply an interface for a *simple* module (**module** ... **end**), then the type checker infers a fully transparent interface by inspecting the module's implementation. For example, under my approach, `StringOrd`, which we bound to the result of evaluating **module type** `T = string;` **type** `cmp = ...;` **end**, would have been assigned the following interface:

interface type `T = string;` **val** `cmp: T*T -> int;` **end**

If we compare this interface to the original opaque `ORDERED` interface (repeated below), we see that the new interface differs only in the addition of the type identity `T=string`:

interface type `T;` **val** `cmp: T*T -> int;` **end**

By using a shorthand notation due to Xavier Leroy [31], we can take advantage of this fact to express the new interface as `ORDERED` **with** `T=string`. Leroy's with notation is a purely syntactic shorthand that expresses the result of adding information about type identities to an existing interface. A discussion of exactly how the with notation works can be found in Subsection 13.7.4.

The situation for non-simple module expressions without interfaces is more complicated, particularly because modules under my approach are treated as first-class values. Briefly, by the use of some type rules, an interface that is as transparent as possible given soundness and reasonable compile-time–information constraints[1] is inferred. I shall give an example of how this can occur later in Section 3.7. Code that does not involve using modules as first-class values, *higher-order functors* (functors that take or return other functors as arguments), or type coercions will be given a fully transparent interface [34]. This category includes the majority of code written in SML.

[1]In particular, the type checker makes no attempt to determine the run-time branching of conditionals, instead assuming that they could branch either way at any time.

3.3 Subtyping

Subtyping in my system permits forgetting type identities; for example, `ORDERED` **with** `T=int` is a subtype of `ORDERED`. This fact means that the programmer can use type coercions (M `<:` **interface** . . . **end**) to increase opacity where and when desired. For example, the following code creates an initially transparent `HashTable` module then makes opaque just the `T` type component:

```
HashTable = (module type elem = int;
                    type T = (string*elem) array;
                    ...
            end
         <: interface type elem = int;
                     type T;
                     ...
           end);
```

This technique provides module-level data abstraction under my approach.

Thus, in summary, under the translucent approach, the default is full transparency; opacity results either from using modules in a first-class manner (to prevent unsoundness) or from using an explicit type coercion (to obtain data abstraction).

3.4 The B-Tree Example

To get a feel for how translucency works, let us consider again the B-Tree example from Section 1.4. What happens if we try to build a telephone directory using `MkBTree` under the translucent approach? If we examine the original code for `MkBTree` (partially copied below), we see that it assigns the interface `ORDERED_DICTIONARY` to the result of applying `MkBTree`:

```
MkBTree =
  functor (K:ORDERED, E:interface type T; end)
             :ORDERED_DICTIONARY
    module
      mod Key = K;
      type elem = E.T;
      type T =  btree(Key.T,elem);
      ...
    end;
```

Because `ORDERED_DICTIONARY` is completely opaque (it specifies no type identities), this means that `MkBTree`'s boundary under the translucent approach will be completely opaque as well; consequently, `MkBTree` will act the same as it did under the opaque approach: If we try to build a telephone directory like we did before in Section 2.1, abstract operation-argument types will result in type errors that prevent us from using the resulting telephone directory.

What happens if we remove the explicit result interface (`:ORDERED_DIRECTORY`) from the `MkBTree` code? In that case, the type checker will infer the following totally-transparent result interface:

```
ORDERED_DICTIONARY with Key.T = K.T
                   with elem  = E.T
                   with T     = btree(K.T,E.T)
```

(Note that the body of the `MkBTree` functor is a simple module.)

With this interface, `MkBTree`'s boundary will be completely transparent; consequently, this version of `MkBTree` will act the same as the old version of `MkBTree` did under the transparent approach: If we try to build a telephone directory, the resulting dictionary will be fully usable, but its implementation will be exposed to the user.

Both versions of `MkBTree` are unsatisfactory; what we really need is for `Key.T` and `elem` to be transparent (so that the resulting dictionary will be usable) and for `T` to be opaque (so that `MkBTree`'s implementation is hidden). We can achieve this by assigning the following translucent interface as the return interface for `MkBTree`:

```
ORDERED_DICTIONARY with Key.T = K.T
                   with elem  = E.T
```

This interface gives `MkBTree` a translucent boundary that is transparent for the `Key.T` and `elem` type components but opaque for the `T` type component.

What happens if we try and create a telephone directory using this new version of `MkBTree` as before (code repeated below)?

```
D = MkBTree(StringOrd, module type T=string; end);

directory = D.new();

D.define(directory, "John Smith", "123-4567");
```

Because of the translucent boundary, `D.define` will have the type `D.T * string * string -> unit`, where `D.T` is an abstract type. Accordingly, there is no problem using the resulting telephone directory and the implementation of `MkBTree` is hidden as desired. Note that only my approach is thus able to handle the B-Tree example in a satisfactory manner.

3.5 The Recipe

My approach is based on type theory and the λ-calculus. In it, modules are a particular sort of value and interfaces are a particular sort of type. This choice yields a simple and uniform design.

My "recipe" for building translucent higher-module systems is as follows:

- Start with a base λ-calculus (to model the core language); I use Girard's F_ω [15] in my Kernel system
- Add *translucent sums* (to module modules)
- Add dependent functions (to model functors); *dependent* functions are needed because the result type of a functor can depend on its argument
- Add a notion of subtyping (to model implementation-interface matching)

The key component here is the new type theoretical construct I call a translucent sum; the other components of the recipe are, with the exception of one minor change to the behavior of dependent functions (see the next chapter), standard components drawn from the extensive type-theory literature.

Indeed, variations of this recipe can be used to build opaque or transparent higher-order module systems: To get an opaque system, for example, weak sums (also called existentials) with the dot notation elimination form (discussed in the next chapter) should be substituted for translucent sums in the recipe. To get a transparent system, on the other hand, strong sums should be substituted instead. It will also be necessary in the transparent case to stratify the system, separating modules, functors, and their "types" from ordinary terms and types; this stratification ensures soundness at the cost of making modules and functors second-class values.

For both of these variations, in order to handle submodules, dependent sums should either be added in addition or their functionality should be merged into the weak or opaque sum construct. (Translucent sums already include the functionality of dependent sums.)

3.6 Translucent Sums

Translucent sums are weak sums with the dot elimination form that have been enriched in the following ways:

- The ability to give equational information about type components has been added
- Type equality has been modified to take this equational information into account

- They have been generalized to:
  - have any number of components
  - have named components
  - allow dependencies on previous (value) components
- Subtyping has been modified to allow
  - forgetting equational information about type components
  - reordering components
  - dropping components

The last two subtyping changes are to allow more flexible matching of module implementations to interfaces. This ability is very useful in practice because it allows modules to be combined in more flexible ways. In particular, it allows passing a module with extra or differently ordered components than a functor's argument interface requires to that functor without having to manually write a coercion function to extract only the required components. The reordering ability also frees programmers of the need to remember the exact order of an interface's components.

I shall discuss translucent sums in more detail in the next chapter.

3.7 First-Class Values

Like weak sums, on which translucent sums are based, translucent sums are ordinary values. This fact has a number of consequences. First, because modules are translucent sums, modules under my approach are first-class values. Second, because of this fact, modules may be nested inside other modules as value components (i.e., using the **val** keyword); my approach thus supports submodules without needing to handle submodule components specially.

Third, because having modules as first-class values means that any function can take or return modules, functors are simply ordinary (dependent) functions; accordingly, functors in my approach are also just ordinary values. Finally, fourth, because functions in λ-calculi are higher-order (they can take and return functions) and functors are just ordinary functions, my approach supports higher-order functors.

The recipe I have described builds a module system where modules are first-class values. It is possible to alter the recipe (see Section 13.8) to build instead a translucent higher-order module system where modules are second-class values. Such an approach would yield a system with the same abilities as the approach I am describing here except that it would not be possible to choose ADT implementations at runtime. The resulting system, however, would require substantially less complicated proofs.

Code using modules as first-class values may automatically lose type identity information in order to avoid unsoundness. Consider again, for example, the code to select a dictionary at runtime based on the number of items needed:

```
Dictionary = if n<20 then LinkedList else HashTable;
```

Suppose the programmer gave fully transparent interfaces to `LinkedList`, say `DICTIONARY` **with** `T=list(...)`, and `HashTable`, say `DICTIONARY` **with** `T = array(...)`. Because the type checker does not know what `n`'s value will be at runtime, it cannot safely determine `Dictionary.T`'s type identity. Accordingly, it must be conservative and assign `Dictionary` the more opaque interface `DICTIONARY`, which makes `Dictionary.T` abstract.

No special type rules are needed to implement this behavior: Because the type rule for if–then–else requires the then and else branches to have the same type and `LinkedList` and `HashTable` have different types, the type checker is forced to use subsumption to coerce both branches to a common supertype; all such types make `Dictionary.T` opaque. It is because of this ability to make types abstract when they cannot be determined at compile-time that my approach is able to treat modules as first-class values without risking unsoundness.

3.8 Linking

One advantage of a programming language with a higher-order module system based on type theory is that the process of separate compilation, including linking, can be described in the language without any need for additional constructs or type rules. The basic idea here is to describe separate compilation as a preprocessing stage that automatically converts files containing code with references to other modules into closed functors by using the files containing the interfaces for the referenced modules. The closed functors can then be compiled completely independently of any other files. The linker must then automatically generate code to apply the compiled functors generated by the compiler to each other in the correct order at runtime.

In order to give you a favor for how this process might work, I am now going to describe a very simple separate-compilation system. In this system, the implementation code for modules named M is placed in the file M`.m` while the interface for module M (only one interface per module is allowed in this system) is placed in the file M`.i`; this naming convention allows the preprocessor to find the interfaces of referenced modules.

Interface and implementation files in this system may optionally start with an import declaration. Only modules listed in a file's import declaration, if any, may be referenced

```
File contents of "String.i":
  interface
    type T;
    val  empty : T;
    ...
  end;

File contents of "String.m":
  module
    type T = array(char);
    val  empty = ...
    ...
  end;
```

Figure 3.1: Code for the `String` module

from that file. If a module's interface file includes an import declaration, then it's implementation file must include an import declaration which starts with the same list of modules imported in the module's interface file.

For an example program, consider Figures 3.1 and 3.2. Figure 3.1 contains the interface and implementation files for a module `String` that implements a string type from character arrays. Figure 3.2 contains the interface and implementation files for a module `Hash`, which implements an integer hash-table ADT keyed on strings using an associative list ADT. The modules `String` and `Assoc` (not shown) have no import declarations because they do not need to refer to any other modules. The `Hash` interface imports `String` (in order to use the type `String.T`), while the `Hash` implementation imports `Assoc` as well (in order to use the `String` and `Assoc` operations to implement hash tables).

The transform of `Hash` into a closed functor results in the code of Figure 3.3. This transform can be performed completely mechanically: the list of functor arguments comes from the implementation's import list, the interfaces from the appropriate interface files, and the functor body from the module's implementation file. Note that no references to external modules remain: The references to `String.T`, `Assoc.T`, and so on, refer to the `MkHash` functor's arguments, not to the external modules of the same name. Also note that checking that `Hash`'s implementation satisfies its publicly declared interface will be done automatically when the transformed code is compiled; the matching relation here is just subtyping on interfaces, avoiding the need for a separate set of rules. The transforms of `String` and `Assoc` (not shown) are much simpler since they import nothing.

When this program is linked together, the linker automatically generates the series

```
File contents of "Hash.i":
  import String;

  interface
    type T;
    val  new : unit -> T;
    val  lookup : T * String.T -> int;
    ...
  end;

File contents of "Hash.m":
  import String, Assoc;

  module
    type T = Assoc.T(int)
    fun  new() = ...
    fun  lookup(table:T, key:String.T) = ...;
    ...
  end;
```

Figure 3.2: Code for the `Hash` module

```
MkHash =
  functor (String: interface
                     type T;
                     val  empty : T;
                     ...
                   end,
           Assoc:  interface ... end)
             : interface
                 type T;
                 val  new : unit -> T;
                 val  lookup : T * String.T -> int;
                 ...
               end
     module
        type T = Assoc.T(int);
        fun  new() = ...
        fun  lookup(table:T, key:String.T) = ...;
        ...
     end;
```

Figure 3.3: Code for the transformed `Hash` module

```
String = MkString();

Assoc  = MkAssoc();

Hash   = MkHash(String, Assoc);
```

Figure 3.4: Code to combine the closed functors

of functor applications shown in Figure 3.4 to combine the closed functors together into a complete program. Determining what order to link the closed functors in is somewhat tricky because of dependencies between modules. In the simple system I am describing, the order is determined by first building a directed graph, whose nodes are the modules and whose edges represent dependencies between modules: An arrow is drawn from M to N iff M imports N or M appears before N in any import list. (The second part gives the programmer some control over the order of module initialization; this control can be important when side effects are involved.) Second, the graph is checked for cycles; it is an error, aborting linking, if any cycles are found. Otherwise, third, a topological sort is done to find a ordering that respects all the dependencies.

At runtime, this series of applications is done as the effect of the program. In a more realistic example, there would be a `Main` module, executed last, whose application would perform the desired computation. Note that if functors are used in the original program, then the transformed code will involve higher-order functors.

3.9 Type-Sharing Specifications

Standard ML has a feature that is sometimes useful called *type sharing*, which is used in interfaces to require that (sub)-component types are equal. To see an example of where this feature might be useful, imagine that we had a number of different ordered-dictionary implementations that we wished to test to make sure they worked correctly. One easy test we could do is to write a functor that takes in two ordered dictionaries and performs random operations on the two dictionaries in parallel; if their answers differ, then at least one of them must be incorrect.

In order to make this functor work, we are going to need to require that the two ordered dictionaries we are passed have the same key and element types. (Otherwise, we will not be able to perform the same operations on both ordered dictionaries.) This requirement can be expressed in SML by using type-sharing specifications:

```
functor RandomTest(structure x:ORDERED_DICTIONARY;
                   structure y:ORDERED_DICTIONARY;
                   ...
                     sharing type x.Key.T = y.Key.T
                     sharing type x.elem  = y.elem
                  ) = ...
```

(SML syntax is slightly different from my notation.)

This SML code can be translated into my approach by using a series of with expressions to set the required-equal type components equal to each other:

```
RandomTest =
  functor (x:ORDERED_DICTIONARY,
           y:ORDERED_DICTIONARY
               with Key.T = x.Key.T
               with elem  = x.elem,
           ...)
  ...
```

This basic idea — picking the type in an specified equivalence class with maximal scope and then setting the other types in that class equal to it using with statements — can used to translate any type-sharing specification from SML into my approach.

3.10 Summary

Summarizing, under my approach the identity of type components start out fully visible; this visibility can later be obscured either automatically when modules are used in a first-class manner in order to ensure soundness, or manually when the programmer inserts a type coercion in order to produce abstraction. This controlled visibility permits data abstraction where desired and modules as first-class values without limiting the uses of higher-order features that depend on modules "containing" other types. My approach also supports separate compilation because modules depend only on interfaces.

Since my approach gives the programmer control over the degree of visibility, she can choose opacity or transparency as needed, getting the best of both worlds. Indeed, because translucency permits the programmer to mix opacity and transparency in the same module (e.g., the interface for `HashTable`), new uses of higher-order features become possible (e.g., the B-Tree example). My approach also provides a number of other benefits:

- Functors are first-class
- Higher-order functors are provided
- Type-sharing specifications can be encoded
- Modules may export type abbreviations
- Linking and separate compilation can be described in the resulting programming language
- The resulting system has a simple and uniform type theory

Chapter 4

Technical Machinery

Now that I've described the basic ideas of my approach, I shall introduce in this chapter the technical machinery behind my approach. In particular, I shall explain the syntax and rules for translucent sums in some detail, describe how the various rules interact to obtain desirable properties such as decidable type equality, and motivate some restrictions in the rules. This chapter is intended to serve as an introduction to the machinery worked out in complete detail in the next part of this dissertation; accordingly, I ignore here a number of uninteresting minor technical issues such as the need for side conditions on many rules in order to avoid variable capture and duplicate field names.

4.1 Dot Notation

Traditionally, in order to make use of the contents of a weak-sum value (also called an existential value), a programmer must first *open* it. If the term M_1 evaluates to a weak sum of type **interface type** `T`; **val** `v`:A; **end**, then it may be opened for use in M_2 by writing **open** M_1 **as** `U,x` **in** M_2 **end**. During M_2's execution, `U` will be bound to the type component of the weak sum and `x` will be bound to its value component, which will have type [`U`/`T`]A. The type rule for `open` requires that M_2's type does not refer to `U`; this requirement is necessary to avoid unsoundness. (Because weak sums are first-class values, `U` may be bound to different types at runtime, making it unsafe to refer to it by a single static name outside the scope of the open statement.)

Having to write out an open statement each time you want to use a weak sum can be inconvenient; in a 1990 paper [11], Luca Cardelli and Xavier Leroy introduced a shortcut they called the dot notation. The basic idea is to allow the programmer to directly write `x.T` for the type component of and `x.v` for the value component of the weak sum denoted by the variable `x`. These direct references are then translated away by placing open statements next to the bindings of the weak sums and by substituting the names of the opened components for the direct references.

For example, consider the following piece of code:

```
let x = $M_1$ in
  x.v.1(λs::x.T. true)
end
```

This code translates into the following piece of code:

```
let x = $M_1$ in
  open x as x_T,x_v in
    x_v.1(λs:x_T. true)
  end
end
```

By using a more complicated translation, Cardelli and Leroy showed, among other things, that their dot notation could be extended from working on just variables to working on any (sub)-component of a variable as well. Under that extension, if `x.y.z`, the `y.z` sub-component of the variable `x`, is a weak sum, then its type component could be referred to directly as `x.y.z.T`. The extension does not compromise soundness because the (sub)-components of variables are essentially just another kind of variable.

I have chosen to build the dot notation into translucent sums directly instead of requiring it to be first translated away or not providing it. In their paper, Cardelli and Leroy do not deal with dependent sums or functions; it seems unlikely that the dot notation can be translated into open statements in the presence of these features, both of which are present in my system. Thus, the dot notation in my system should be regarded as an independent feature and not an abbreviation for the use of open. As I shall explain in Section 4.5, my approach is able to support the dot notation on a somewhat larger class of terms than variables and their (sub)-components.

4.2 Creation

For the this chapter, I shall use the following syntax:

Simple Modules	S	$::=$	**module** B_1; $\cdots$ B_n; **end**	$(n \geq 0)$
Interfaces	I	$::=$	**interface** D_1; $\cdots$ D_n; **end**	$(n \geq 0)$
Bindings	B	$::=$	**val** x=M | **type** α=A	
Declarations	D	$::=$	**val** $x : A$ | **type** α=A | **type** α	
Terms	M	$::=$	x | S | $M.x$ | **functor** $(x : A)$ M | $M_1\, M_2$ | ...	
Types	A	$::=$	α | I | **FUNCTOR** $(x : A) : A'$ | $M.\alpha$ | ...	
Assignments	,	$::=$	$D_1; \cdots D_n$	$(n \geq 0)$

Here, the metavariable α ranges over *type variables* and the metavariable x ranges over *term variables.* I have omitted the syntax for constructs other than translucent sums and dependent functions because it will not be needed; likewise, I have simplified the presentation of translucent sums here by using only one name per field rather than the pair of names — one for internal access and one for external access — that are required for technical reasons (see Section 13.1). In the next chapter I shall introduce a more concise, precise, and formal notation for my system.

In order to type check a simple module, we type check each of its bindings under the preceding bindings, if any, and the current assignment (assignments describe the variables from the rest of the program currently in scope):

$$\frac{\forall i \in [1..n].\ ,\ ; D_1; \cdots ; D_{i-1} \vdash B_i : D_i}{,\ \vdash \textbf{module}\ B_1; \cdots ; B_n;\ \textbf{end} : \textbf{interface}\ D_1; \cdots ; D_n;\ \textbf{end}} \qquad \text{(I-SIMPLE)}$$

The resulting type for the module is an interface type listing the "types" of the module's bindings. The "type" of a binding is a declaration; the following rules associate declarations with bindings:

$$\frac{,\ \vdash M : A}{,\ \vdash \textbf{val}\ x{=}M : \textbf{val}\ x{:}A} \qquad \text{(I-VAL)}$$

$$\frac{,\ \vdash A\ \text{valid}}{,\ \vdash \textbf{type}\ \alpha{=}A : \textbf{type}\ \alpha{=}A} \qquad \text{(I-TYPE)}$$

Here , $\vdash M : A$ denotes that term M has type A under assignment , and , $\vdash A$ valid denotes that A is a valid type under , . Note that the rules assign transparent declarations (**type** $\alpha{=}A$) rather than opaque declarations (**type** α) to type bindings. This behavior implements the default interface policy for simple modules (full transparency) that I described in Section 3.2.

As an example, these rules assign **module type** α`=int;` **val** `a=2;` **val** `b=a+2;` **end** the type **interface type** α`=int;` **val** `a:int;` **val** `b:int;` **end** under , . Note that because we type check successive bindings in the scope of the previous ones, fields can refer to earlier fields. In the above example, the `b` field was able to refer to the `a` field because it was type checked under the assignment , ;**type** α=int;**val** x:int. Earlier type fields such as α may also be referred to in later fields.

Later fields may also refer to (sub)-components of earlier fields. For example, the following module is well typed:

```
module val  P=module
                type T=int;
                val  x=3;
              end;
       type U=P.T;
       val  y=P.x;
end
```

Note the use of the dot notation here to refer directly to the type `P.T`. The rule for determining when such a reference is valid is (roughly) as follows:

$$\frac{,\ \vdash M : \textbf{interface}\ ,_1;\textbf{type}\ \alpha;,_2;\ \textbf{end}}{,\ \vdash M.\alpha\ \text{valid}} \qquad \text{(V-DOT)}$$

That is to say, if M denotes a translucent sum with an α type field[1] under , , then $M.\alpha$ denotes a type. This rule is in fact too general; I shall explain why and how the set of terms that this rule applies to needs to be restricted in Section 4.5. I shall defer discussion of how $M.x$ is typed until Section 4.6.

The ability to refer to earlier fields also exists in interfaces:

$$\frac{\forall i \in [1..n].\ ,\ ; D_1; \cdots ; D_{i-1} \vdash D_i\ \text{valid}}{,\ \vdash \textbf{interface}\ D_1; \cdots ; D_n;\ \textbf{end}\ \text{valid}} \qquad \text{(V-INTER)}$$

Thus, interfaces like **interface type** α; **val** `x`:α; **end** are valid.

All of the rules for dependent functions (functors) are standard except for the typing rule for application, which I shall explain in Section 4.6. For example, the rule for type

[1]Note that a transparent declaration for α in M's interface can always be turned into an opaque declaration for α by using subsumption on M to forget the identity of α.

checking functors is as follows:

$$\frac{,\ ; x{:}A \vdash M : A'}{,\ \vdash \textbf{functor}\ \ (x{:}A)\ \ M : \textbf{FUNCTOR}\ \ (x{:}A){:}A'} \qquad \text{(I-FUNC)}$$

Note that the result type of a functor (A') can depend on its argument (x).

4.3 Equality

Type equality in my system is structural (i.e.., component-wise) except for equations derived from the information about type identities found in translucent sums and from reductions involving functions on types (not discussed in the first part of this dissertation; see Section 5.1). In particular, two interfaces are equal if their corresponding declarations are equal:

$$\frac{\forall i \in [1..n].\ ,\ ; D_1; \cdots ; D_{i-1} \vdash D_i = D_i'}{,\ \vdash \textbf{interface}\ D_1; \cdots ; D_n;\ \textbf{end} = \textbf{interface}\ D_1'; \cdots ; D_n';\ \textbf{end}} \qquad \text{(E-INTER)}$$

Two declarations are equal if they are of the same sort, declare the same variable, and contain equal types:

$$\frac{,\ \vdash A = A'}{,\ \vdash \textbf{val}\ x{:}A = \textbf{val}\ x{:}A'} \qquad \text{(E-VAL)}$$

$$,\ \vdash \textbf{type}\ \alpha = \textbf{type}\ \alpha \qquad \text{(E-TYPE-O)}$$

$$\frac{,\ \vdash A = A'}{,\ \vdash (\textbf{type}\ \alpha{=}A) = (\textbf{type}\ \alpha{=}A')} \qquad \text{(E-TYPE-T)}$$

Note that equality does not permit the reordering of translucent-sum fields.

The information about type identities contained in translucent sums leads to equations in two ways. First, whenever a transparent declaration occurs in an assignment, it gives rise to an equation involving its type variable:

$$,\ ; \textbf{type}\ \alpha{=}A; ,\ ' \vdash \alpha = A \qquad \text{(ABBREV)}$$

Because the components of translucent sums (and functors) are compared for equality under an assignment extended with the previous declarations, if any, this rule give rise to equations like the following:

$$\begin{array}{l} \textbf{interface type}\ \alpha\texttt{=int;}\ \ \textbf{val}\ \texttt{x:}\alpha\texttt{;}\ \ \ \textbf{end}\ \ = \\ \textbf{interface type}\ \alpha\texttt{=int;}\ \ \textbf{val}\ \texttt{x:int;}\ \ \textbf{end} \end{array}$$

Second, declarations of translucent sums containing transparent declarations, either directly or in a sub-component, also give rise to equations:

$$\frac{\begin{array}{c}\mathrm{FV}(A) \cap \mathrm{fields}(,_1) = \emptyset \\ , \vdash M : \textbf{interface } ,_1; \textbf{type } \alpha{=}A; ,_2; \textbf{ end}\end{array}}{, \vdash M.\alpha = A} \qquad (\mathrm{ABBREV}')$$

As with the previous rule involving the dot notation, V-DOT, this rule is too general; a similar restriction to the set of terms it applies to will need to be made.

I use $\mathrm{FV}(A)$ to denote the free variables of A and fields(,) to denote the fields declared in , . Thus, the first precondition of the ABBREV′ rule expresses the requirement that the type α is equal to (A) does not *depend* on any earlier fields in M's type; this precondition is needed to prevent variable capture.

Such dependencies can be classified as either *essential* or *inessential*. A dependency is inessential if it can be removed by the use of the equality rules alone; otherwise, it is essential. Dependencies on transparently-defined fields are inessential: they can be removed by using the ABBREV or the ABBREV′ rule as appropriate to replace the field name with its declared identity. (The typing rules prohibit fields from depending on themselves.) Dependencies on opaquely defined fields, on the other hand, are usually essential.

As an example, in the following interface type the `z` field depends essentially on the α field but only inessentially on the β field:

```
interface
  type α;
  type β = int;
  val  z: FUNCTOR (s:α):β;
end
```

The difference between essential and inessential dependencies is important because several type rules in my system require all dependencies between certain fields to be removed before they can be applied.

Like the ABBREV rule, the ABBREV′ rule also gives rise to equations on interfaces when combined with the component-wise equality rules. For example, we have that

```
interface
  val  x: interface type T=int; end;
  type U = x.T;
end
```

is equal to

```
interface
  val  x: interface type T=int; end;
  type U = int;
end
```

4.4 Subtyping

Subtyping in my system is structural modulo equality — equal types are always subtypes of each other — except for forgetting information about type-component identities and the reordering and dropping of translucent-sum fields. The component-wise subtyping rule for translucent sums says that two interfaces are subtypes if their corresponding declarations are "subtypes":

$$\frac{\forall i \in [1..n].\ ,\ ; D_1; \cdots ; D_{i-1} \vdash D_i \leq D'_i \qquad ,\ \vdash \textbf{interface}\ D'_1; \cdots ; D'_n;\ \textbf{end}\ \text{valid}}{,\ \vdash \textbf{interface}\ D_1; \cdots ; D_n;\ \textbf{end} \leq \textbf{interface}\ D'_1; \cdots ; D'_n;\ \textbf{end}} \quad \text{(S-INTER)}$$

One declaration is a structural "subtype" of another if it is of the same sort and declares the same variable either to be a subtype of the other declaration's declared type (value-declaration case) or to be equal to the same type(s) modulo equality (type-declaration case):

$$\frac{,\ \vdash A \leq A'}{,\ \vdash \textbf{val}\ x{:}A \leq \textbf{val}\ x{:}A'} \quad \text{(S-VAL)}$$

$$,\ \vdash \textbf{type}\ \alpha \leq \textbf{type}\ \alpha \quad \text{(S-TYPE-O)}$$

$$\frac{,\ \vdash A = A'}{,\ \vdash (\textbf{type}\ \alpha{=}A) \leq (\textbf{type}\ \alpha{=}A')} \quad \text{(S-TYPE-T)}$$

Note that it is not safe to replace a transparent type declaration with one that declares the field name to be equal to a subtype of the original type because type fields can be used in contravariant positions; types in such positions can in general only be replaced safely with supertypes.

The forgetting of type-identity information is implemented by a non-structural rule for declaration "subtyping":

$$\frac{,\ \vdash A\ \text{valid}}{,\ \vdash (\textbf{type}\ \alpha{=}A) \leq (\textbf{type}\ \alpha)} \quad \text{(S-TYPE-F)}$$

Some examples of subtyping using these rules are as follows:

```
(a)      interface type T=int; val x:int; end
       ≤ interface type T=int; val x:T;   end
       ≤ interface type T;     val x:T;   end

(b)      interface type T; type U=T; end
       ≤ interface type T; type U;   end
```

Here in (a), we first created an inessential dependency of the `x` field on `T` using equality then converted it into an essential dependency by using forgetting to make `T` an opaque field. In (b), by contrast, we broke an essential dependency of the `U` field on `T` by forgetting that `U` was equal to `T`. Subtyping, thus, unlike equality, can be used to break or create essential dependencies.

Note that before a field's type-identity information can be forgotten, it may be necessary to remove some dependencies on that field in order to ensure that the resulting type is valid as required by the second precondition of rule S-INTER. (Such dependencies can always be removed by substituting the type the field is declared equal to for the field name everywhere.) Consider the following valid interface:

```
interface
  type S = interface type T; end;
  val  M:S;
  val  N:S;
  val  x:M.T;
end
```

We cannot directly forget `S`'s identity because the resulting type is invalid:

```
interface
  type S;
  val  M:S;
  val  N:S;
  val  x:M.T;
end
```

(Here, `M.T` is an invalid type because `M`, being of a (now) abstract type, cannot be shown to be a translucent sum.)

However, we can forget `S`'s identity if we first replace the occurrence of `S` in `M`'s type with the interface it is currently defined equal to, resulting in the following type:

```
interface
  type S;
  val  M : interface type T; end;
  val  N:S;
  val  x:M.T;
end
```

In addition to the component-wise subtyping rule for interfaces, there are additional non-structural rules that permit the reordering and dropping of translucent-sum fields. Because it is unclear how the rules that implement this behavior should be formulated (see Section 13.2), I shall just give a very rough description of their behavior here: Fields in interfaces may be dropped if no other fields (essentially) depend on them. A field may be moved to the right past other fields if they do not depend (essentially) on the moving field. Some variants of these rules allow a field to be moved to the right past fields that depend essentially on the moving field under some conditions. See Section 13.2 for details.

Consider, for example, the following interface:

```
interface type a; type b=int; val c:b; val d:a; end
```

Here, the `c` and `d` fields could be dropped directly and the `b` field could be dropped after the dependency on it by the `c` field had been broken by using equality. The `a` field, by contrast, cannot be dropped unless the `d` field is dropped first because the `d` field depends on it essentially. The `a` field could be moved right past the `b` and `c` fields but not past the `d` field. Likewise, the `b` field could be moved right past the `c` and `d` fields only if the dependency by `c` on `b` was first broken.

Note that, like the rule for ordinary non-dependent functions, the subtyping rule for functor types is contravariant:

$$\frac{,\ \vdash A_2 \leq A_1 \qquad ,\ ; x{:}A_2 \vdash A_1' \leq A_2'}{,\ \vdash \textbf{FUNCTOR}\ (x : A_1) : A_1' \leq \textbf{FUNCTOR}\ (x : A_2) : A_2'} \qquad \text{(S-FUNC)}$$

Here, the direction of subtyping on the functor argument types (A_i) is reversed from that on the overall types. This fact means that subtyping can be used to "remember" the type-identity information of variables declared in contravariant positions. This ability provides another way for dependencies to be broken:

```
   FUNCTOR (x:interface type T;     end):x.T
≤  FUNCTOR (x:interface type T=int; end):x.T
≤  FUNCTOR (x:interface type T=int; end):int
```

Here, remembering followed by equality was used to break the essential dependency of the result type on the argument `x`.

4.5 The Phase Distinction

One desirable property of a programming language is that it have a *phase distinction* [22]. A programming language is said to have a phase distinction if programs in it can be type checked without evaluating general program expressions (terms). Programming languages without a phase distinction are hard to compile efficiently; in the presence of side effects, their semantics are also unclear. (What does it mean for a program's type to be "if today is Monday then `int` else `bool`"?)

Surprisingly, my system has a phase distinction in spite of allowing modules as first-class values. The only difficult part to ensuring a phase distinction for my system is figuring out how to handle equality on types containing terms. In particular, how should equations like the following be checked?

$$M.\alpha = N.\alpha$$

The naive method is to simply evaluate M and N, read off the types of their α type components from the resulting values, and then compare the types using equality. However, since M and N may be arbitrary terms, this rules out having a phase distinction.

My answer is to restrict the set of terms that may occur in types to an extended set of "values" I call *extended values*. The set of extended values consists of the usual call-by-value values (denoted by the metavariable V) extended with term variables and selections on extended values:

Definition 4.5.1 (Syntax)
Extended Values ν ::= $x \mid V \mid \nu.y$

I choose this set because it allows referring to any previous type component (including nested cases like `x.y.`α) and is closed under substitution of a call-by-value value for a term variable, making defining the call-by-value semantics of function application much easier. Note that type coercions of the form V<:A are not considered call-by-value values because they can be reduced further to V. If they were added to the set of extended values, then the identity of types like (**module type** `T=int;` **end** `<:` **interface type** `T;` **end**)`.T` becomes problematic because in order for abstraction to work properly, this type should not even be equal to itself, let alone be equal to `int`.

This restriction, which I call the extended-value restriction, is implemented both syntactically, by replacing $M.\alpha$ in the grammar for types with $\nu.\alpha$, and via the type rules, by replacing M by ν in the rules V-DOT and ABBREV′. The restriction makes computing the identity of a type component like $\nu.\alpha$ easy: The only operation that needs to be performed is selection, which never produces side effects and cannot cause non-termination.

The computing of type identities for $\nu.\alpha$'s is done in my system by the ABBREV$'$ rule combined with the rules for typing simple modules and selections. These rules give rise to equations like the following:

```
(module type T=int; val x=3; end).T = int
(module type T=int; type U=T*bool; end).U = int*bool
(module val x = module type T=int; end; end.x).T = int
```

Using these equations, any valid type in my system can be converted using equality into an equal type that contains terms of only the form $x.y_1.y_2\cdots.y_n$, $n \geq 0$. I shall call such terms *places*. (The valid-type requirement here is to rule out cases with invalid selections like $(\lambda x{:}\texttt{int}.\,3).\alpha$.) Once the need for evaluating terms has been removed, it is easy to establish a phase distinction.

Note that the remaining terms are of the form that Cardelli and Leroy showed how to convert into open statements (see Section 4.1); this fact indicates that my extension of the dot notation to extended values is conservative because $\nu.\alpha$'s can be regarded as just syntactic sugar for a type using only the unextended dot notation.

One consequence of the extended-value restriction is that my system is limited to call-by-value: Call-by-name requires the ability to substitute general terms for term variables in terms, which in turn requires the ability to substitute general terms for term variables in types because terms may contain types; however, substituting general terms into types results in types containing general terms in violation of the restriction.

Attempting to relax the extended-value restriction in order to get around this limitation gives unsoundness in the presence of side effects. To see this, consider the following example, where the `alternate` function alternates between returning `true` and `false` (alternate uses side effects internally):

```
IntPkg  = module type T = int;
                 val  v = 3;
                 val  f = negate;
          end;

BoolPkg = module type T = bool;
                 val  v = true;
                 val  f = not;
          end;
```

```
Apply   = functor (Pkg: interface type T;
                                 val   v:T;
                                 val   f:FUNCTOR (x:T):T;
                            end)
              begin
                Pkg.f(Pkg.v);
                true;
              end;

Apply(if alternate() then IntPkg else BoolPkg)
```

This code is well typed in my system[2] and safe under call-by-value because `Apply` under call-by-value applies the function `f` to the value v from the same package. However, if we evaluate this code under call-by-name, we evaluate `Apply`'s argument twice resulting in a type error because we end up evaluating either `IntPkg.f(BoolPkg.v) = negate(true)` or `BoolPkg.f(IntPkg.v) = not(3)`. A similar runtime type error occurs if we first β-reduce or inline the call to `Apply`, then evaluate using call-by-value. Because the β-expansion would type check if the extended-value restriction were relaxed, this example shows that such a relaxation is unsound in the presence of effects.

4.6 Selection and Application

Another consequence of the extended-value restriction is that the traditional elimination rules[3] for dependent sums and functions cannot be used because they require the ability to substitute general terms into types:

$$\frac{, \vdash M : \textbf{interface val } \texttt{x}{:}A_1;\ \textbf{val } \texttt{y}{:}A_2;\ \textbf{end}}{, \vdash M.\texttt{x} : A_1} \qquad \text{(I-TRAD-S1)}$$

$$\frac{, \vdash M : \textbf{interface val } \texttt{x}{:}A_1;\ \textbf{val } \texttt{y}{:}A_2;\ \textbf{end}}{, \vdash M.\texttt{y} : [M.\texttt{x}/\texttt{x}]A_2} \qquad \text{(I-TRAD-S2)}$$

$$\frac{, \vdash M_1 : \textbf{FUNCTOR } (x{:}A){:}A' \qquad , \vdash M_2 : A}{, \vdash M_1\,M_2 : [M_2/x]A'} \qquad \text{(I-TRAD-F)}$$

[2]I have used an imperative version of `Apply` here, which executes `Pkg.f(Pkg.v)` solely for its side effects, in order to avoid a restriction on functor applications that I discuss in the next section.

[3]Elimination rules for a construct handle type checking that construct's elimination form(s), in this case selection for dependent sums and application for dependent functions.

Here, I have simplified things by only giving the case for binary dependent sums and have used $[M/x]A$ to denote the substitution of the term M for the term variable x in the type A in a variable capture avoiding way.

In order to avoid this problem, I limit my selection and application rules to the non-dependent cases, which do not require term substitution to handle:

$$\frac{, \vdash \nu : \textbf{interface } ,_1; \textbf{type } \alpha; ,_2; \ \textbf{end}}{, \vdash \nu.\alpha \text{ valid}} \qquad \text{(V-DOT)}$$

$$\frac{\begin{array}{c}\text{FV}(A) \cap \text{fields}(,_1) = \emptyset \\ , \vdash \nu : \textbf{interface } ,_1; \textbf{type } \alpha{=}A; ,_2; \ \textbf{end}\end{array}}{, \vdash \nu.\alpha = A} \qquad \text{(ABBREV}'\text{)}$$

$$\frac{\begin{array}{c}\text{FV}(A) \cap \text{fields}(,_1) = \emptyset \\ , \vdash M : \textbf{interface } ,_1; \textbf{val } x{:}A; ,_2; \ \textbf{end}\end{array}}{, \vdash M.x : A} \qquad \text{(I-SELECT)}$$

$$\frac{\begin{array}{c}, \vdash M_2 : A \\ , \vdash M_1 : \textbf{FUNCTOR } (x{:}A){:}A' \qquad x \notin \text{FV}(A')\end{array}}{, \vdash M_1\,M_2 : A'} \qquad \text{(I-APP)}$$

When can these rules be applied if we assume that all of their preconditions except the no-dependency one(s) are met? Clearly, the V-DOT rule can always be applied because opaque declarations cannot depend on anything. Less obviously, if M can be given a *fully-defined transparent type* — a type that declares all of M's type (sub)-components transparently — that also satisfies the traditional preconditions then the I-SELECT rule can be applied to M and, furthermore, if M is an extended value, then the ABBREV$'$ rule can be applied to it as well. This fact holds because all dependencies within and on a fully-defined transparent type must be inessential and hence breakable via subsumption; moreover, this breaking can be done without invalidating the traditional preconditions.

The case for the I-APP rule is similar but the reasoning is more complicated. Here, it is the argument M_2 that needs to be given a fully-defined transparent type in order for the rule to apply. For the traditional preconditions to hold, this type must be a subtype of M_1's domain type. Given these two conditions, the I-APP rule can always be applied: first use subtyping to specialize the functor M_1's domain type to the fully-defined transparent type for M_2; second, use equality to break all dependencies by the functor result type (A') on the functor argument (x), which will now all be inessential; and third, apply the I-APP rule to the resulting non-dependent function type.

As an example, consider type checking the application of a functional version of `Apply` (`FApply`, defined below) to `IntPkg` (defined in the previous section).

```
FApply  = functor (Pkg: interface type T;
                                  val   v:T;
                                  val   f:FUNCTOR (x:T):T;
                              end)
              Pkg.f(Pkg.v);
```

FApply and IntPkg have the following types:

```
IntPkg : interface type T=int;
                   val v:int;
                   val f:FUNCTOR (x:int):int;
         end

FApply : FUNCTOR (Pkg: interface
                         type T;
                         val   v:T;
                         val   f:FUNCTOR (x:T):T;
                       end):Pkg.T
```

Note that IntPkg's type is a fully-defined transparent subtype of FApply's domain type (use equality to replace occurrences of int with T then forget the type identity of T) so the functor application will be well typed if we can remove the dependency on Pkg by FApply's result type.

By using subtyping via subsumption, we can specialize FApply's domain type to IntPkg's type:

```
FApply : FUNCTOR (Pkg: interface
                         type T=int;
                         val   v:int;
                         val   f:FUNCTOR (x:int):int;
                       end):Pkg.T
```

Next, we can use equality via subsumption to replace all occurrences of Pkg.T in FApply's result type by int; this step breaks the dependency on FApply's domain type by its result type:

```
FApply : FUNCTOR (Pkg: interface
                         type T=int;
                         val   v:int;
                         val   f:FUNCTOR (x:int):int;
                       end):int
```

Finally, we can apply the resulting non-dependent version of `FApply` to `IntPkg` using I-APP to get that `FApply(IntPkg)` has type `int`. Note that we cannot type `FApply(IntPkg)` by first using subsumption to coerce M_2's type to that of `FApply`'s domain type and then using I-APP because once the information about `IntPkg.T` is lost there is no way to break the dependency on `Pkg.T` by `FApply`'s result type.

Because any term that does not denote a translucent sum will have a fully-defined transparent type (because it will have no type (sub)-components), the above result means that the application of a functor to any such term will never fail due to the no-dependency precondition. Thus, functors act like standard dependent functions when applied to anything other than a translucent sum.

There are two special cases worthy of note where translucent-sum expressions can always be given fully-defined transparent types. The first such case is translucent sums that contain no type (sub)-components. Such translucent sums are essentially ordinary records; their types are always fully-defined transparent types with no dependencies between fields because they contain no type (sub)-components, which could be depended on.

The second case concerns translucent-sum extended values; in the next section I shall introduce two rules I call the EVALUE rules, which allow a fully-defined transparent type to be given to any extended value without losing any information about its fields. This ability means that $\nu.\alpha$, $\nu.x$, and $F(\nu)$ will always type check if ν has an α type field, ν has an x type field, or ν is a member of F's domain type, respectively. Because the ABBREV$'$ rule is restricted to working on only extended values, this fact means that the no-dependency restriction does not in fact restrict the occasions when the ABBREV$'$ rule can be applied. Note also that module names (e.g., `IntPkg` or `SearchTree.Binary`) are special cases of extended values; hence, selection of and application to a named module cannot fail due to the no-dependency restriction.

What about the remaining cases where the no-dependency restriction cannot be met? By the process of elimination, these are cases where we are computing a new module (rather than referring to a previously computed module by name), which contains at least one type component that is depended on by the resulting type and whose identity the type checker does not know. Recall that unknown type components can only occur when the programmer either uses modules in a first-class manner or uses a type coercion to intentionally forget the identity of a type component.

These remaining cases need to be blocked in order to avoid unsoundness. The intuition behind this fact is that in these cases the result type of the expression in question contains a type that we have no name for; because we have no name for that type, we cannot be sure that it does not vary at runtime, leading to possible unsoundness. (Because my system uses call-by-value, type names must denote a constant type in each of their dynamic scopes.) As an example, consider the following unsafe code:

```
(if alternate() then IntPkg else BoolPkg).f(
  (if alternate() then IntPkg else BoolPkg).v)
```

This is the β-reduction of the unsoundness example from Section 4.5. Note that it would be well typed if we removed the no-dependency restriction on selection and used the traditional selection rules for dependent sums. This example can be η-expanded to produce an similar unsoundness example for functor application:

```
SelectF = functor (Pkg: interface type T;
                                  val   v:T;
                                  val   f:FUNCTOR (x:T):T;
                             end)
            Pkg.f;

SelectV = functor (Pkg: interface type T;
                                  val   v:T;
                                  val   f:FUNCTOR (x:T):T;
                             end)
            Pkg.v;

SelectF(if alternate() then IntPkg else BoolPkg)(
  SelectV(if alternate() then IntPkg else BoolPkg))
```

Note that this example does not violate the no-dependency restriction on selection because selection is applied only to module names and that both `SelectF` and `SelectV` are well typed even in the presence of the no-dependency restriction.

4.7 The EVALUE rules

Suppose the current assignment contains the following declaration:

val `P` : **interface type** `T;` **val** `x:T;` **end**

What types can we give to the expression `P` under this assignment? Because we have a name, `P`, for the translucent-sum expression we are trying to type, we have a name for the contents of its `T` field, namely `P.T`. This suggests that we can give `P` the type **interface type** `T=P.T;` **val** `x:P.T;` **end**, which is a subtype of the declared type for `P`.

This technique of giving a more expressive type to translucent sums when we have a name for their type components can be generalized to work on arbitrary translucent-sum extended values. The name in this case is simply ν`.T` where ν is the extended value in question. The technique cannot be extended to general terms because of the extended-value restriction (M.T is not in general a valid type in my system). The following two typing rules implement this technique:

$$\frac{, \vdash \nu : \textbf{interface}\ ,_1;\ \textbf{type}\ \alpha;\ ,_2;\ \textbf{end}}{, \vdash \nu : \textbf{interface}\ ,_1;\ \textbf{type}\ \alpha\texttt{=}\nu.\alpha;\ ,_2;\ \textbf{end}} \qquad \text{(EVALUE-T)}$$

$$\frac{\begin{array}{c}, \vdash \nu.x : A' \\ , \vdash \nu : \textbf{interface}\ ,_1;\ \textbf{val}\ x\texttt{:}A;\ ,_2;\ \textbf{end}\end{array}}{, \vdash \nu : \textbf{interface}\ ,_1;\ \textbf{val}\ x\texttt{'}\texttt{:}A;\ ,_2;\ \textbf{end}} \qquad \text{(EVALUE-V)}$$

(The EVALUE-V rule is used in cases of nested translucent sums to apply the technique recursively.)

We can give any extended value ν a transparent type using these rules by repeating the following sequence of steps as many times as possible: first, find the leftmost type (sub)-component in ν's current type that is not declared transparently, call it C. Second, use equality (via subsumption) to break all dependencies on earlier type (sub)-components in ν's type. (We can do this breaking because all earlier type (sub)-components must be declared transparently by step one.) Third, use the EVALUE rules to convert the C field from an opaque declaration to a transparent one. (Step two here is necessary to ensure that the selection in the EVALUE-V rule does not fail due to the no-dependency restriction.) Note that equality (via subsumption) may be needed before applying the EVALUE rules in order to rewrite ν's type into the form **interface** ... **end**.

If ν's original type was *fully-defined* (it declares all its type (sub)-components), then the resulting type will be a fully-defined transparent type. A fully-defined type can be found for ν by simply type checking ν without reordering or dropping fields from ν and its (sub)-components. This procedure can always be done if ν is well typed (adding extra information does not make well-typed terms ill-typed). Thus, we can give a fully-defined transparent type to any well-typed extended value using the EVALUE rules.

As an example, consider giving a fully-defined transparent type to the extended value x under an assignment that contains the following declaration:

```
val x:interface type T; val y:interface type U; type V=T; end; end
```

The extended value x goes through the following types as we give it a transparent type:

```
interface type T;     val y:interface type U; type V=T;    end; end
interface type T=x.T; val y:interface type U; type V=T;    end; end
interface type T=x.T; val y:interface type U; type V=x.T;  end; end
interface type T=x.T; val y:interface type U=x.y.U;
                                              type V=x.T;  end; end
```

In addition to their interactions with the elimination rules, the EVALUE rules are important because they prevent the identity of opaquely-defined type (sub)-components from being lost when the (sub)-components are given a new name. Without these rules, an abstract type and its copy would be considered different abstract types. Consider, for example, the following code fragment:

```
M =    module  type T=int; type U=int; type V=bool; end
    <: interface type T;     type U=T;   type V=bool; end;

N = M;

O = module val X=N; end;
```

Using the EVALUE rules we can give the following transparent types to these module names:

```
M : interface type T=M.T; type U=M.T; type V=bool; end

N : interface type T=M.T; type U=M.T; type V=bool; end

O : interface
      val X : interface type T = M.T;
                        type U = M.T;
                        type V = bool;
              end;
    end
```

Accordingly, we can infer that `M.T = M.U = N.T = N.U = O.X.T = O.X.U` and that `M.V = N.V = O.X.V = bool`, but not that `M.T = int`. These equations mean that `M`, `N`, and `O.X` are interchangeable. Without the EVALUE rules we would only be able to infer that `M.T = M.U`, `N.T = N.U`, `O.X.T = O.X.U`, and `M.V = N.V = O.X.V = bool`; `M`, `N`, and `O.X` are not interchangeable in that case because they do not contain equivalent `T` components.

Note that the EVALUE rules, the elimination rules for selection and application, and the ABBREV rules working in concert are able to propagate all the needed information about the identity of type (sub)-components solely through types; no term (even just a value) ever needs to be substituted during type checking. This machinery is what allows my system to support separate compilation: All the information necessary for type checking can be placed in a module's type (interface); there is no need to have the code of a supplier module so that it may be substituted.

One way to show the power of this machinery is to show that all the information available from substituting an extended value ν for x can also be made available by giving x an appropriate type A possessed by ν (i.e., a fully-defined transparent type as discussed earlier). This claim can be formalized as the following proposition: If ν is a well-typed extended value under , then there exists a type A such that A is a *minimal* type for ν under , and for all types A' such that $, ; [\nu/x], ' \vdash [\nu/x]A'$ valid, we have that $, ; x{:}A; , ' \vdash [\nu/x]A' = A'$. The type A is a transparent minimal type for M under , if for all types A' such that $, \vdash M : A'$, we have that $, \vdash A \leq A'$. I prove a version of this proposition in Sections 11.5 and 11.6 (see especially Theorem 11.5.10 and Lemma 11.6.10) for my kernel system.

This proposition can be used to show that the following versions of the traditional elimination rules, which have been restricted to extended values, can be derived in my system:

$$\frac{, \vdash \nu : \textbf{interface val } \texttt{x}{:}A_1;\ \textbf{val } \texttt{y}{:}A_2;\ \textbf{end}}{, \vdash \nu.\texttt{y} : [\nu.\texttt{x}/\texttt{x}]A_2} \qquad \text{(I-TRAD-S2')}$$

$$\frac{, \vdash M : \textbf{FUNCTOR } (x{:}A){:}A' \qquad , \vdash \nu : A}{, \vdash M\,\nu : [\nu/x]A'} \qquad \text{(I-TRAD-F')}$$

(For simplicity, I again show only the binary sum case here.)

I shall just sketch the proof of the first rule here: Let ν have the type **interface val** $\texttt{x}{:}A_1$; **val** $\texttt{y}{:}A_2$; **end** under , . By applying the proposition, we can get a transparent minimal type for $\nu.\texttt{x}$, call it A_1'. By EVALUE-V then, ν must also have the type **interface val** $\texttt{x}{:}A_1'$; **val** $\texttt{y}{:}A_2$; **end** under , . By the second part of the proposition then, this type is equal to **interface val** $\texttt{x}{:}A_1'$; **val** $\texttt{y}{:}[\nu.\texttt{x}/\texttt{x}]A_2$; **end**. Note that the `y` field of this type cannot depend on the `x` field because it contains no `x`'s (ν cannot contain `x` because of (unshown) side conditions required to prevent variable capture). Accordingly, we can apply equality (via subsumption) to give ν this type, then I-SELECT to show that $\nu.y$ has the desired type $[\nu.\texttt{x}/\texttt{x}]A_2$.

Thus, for selection from and application to extended values, my elimination rules are equivalent to the traditional ones. A similar result can be shown for the other special cases (non–translucent-sum values and translucent sums without type (sub)-components) mentioned before in Section 4.6. My rules differ in the remaining cases in that they refuse to allow some operations that the traditional rules permit in order to prevent unsoundness in the presence of modules as first-class values and to avoid the need for substituting general terms so as to be able to support separate compilation.

Part II

The Kernel System

Chapter 5

Overview

In order to demonstrate my approach, I now present a calculus based on the idea of translucent sums. I omit features that add only surface complexity without making the development more interesting (e.g., syntactic sugar or array types) in order to simplify the presentation. The system I describe is thus a kernel system, rather than a full programming language. Nonetheless, I believe that it contains all the important elements needed in a programming language with higher-order modules. I include both recursive types (allowing for non-terminating programs) and references (allowing for imperative programming) to demonstrate that translucent sums are compatible with effects.

I make a number of other simplifications so that the kernel system proofs are as simple as possible without losing any results. These simplifications include the fact that translucent-sum expressions in kernel-system types are limited to places ($x.y_1 \cdots .y_n$, $n \geq 0$) and the factoring of translucent sums into two simpler constructs, a standard dependent sum and a new construct called a *reified constructor* (see Section 5.5 for details). The more complicated system I described in the first part of this dissertation can be recovered from the kernel system by adding a pre-processing elaboration stage. I choose to move the problem of reordering and dropping translucent-sum components to this elaboration stage. I do this both to simplify the system and because there is some uncertainly about what the best choice of subtyping rules is for handling component reordering (see Section 13.2).

In addition to giving the syntax, the typing rules, and the semantics for the system, I also give formal proofs of all the important properties for a programming language. In particular, I prove that type checking modulo subtyping is decidable and that the system is sound. Unfortunately, subtyping, and hence full type checking, is only semi-decidable; this is unlikely to cause problems in practice though (see Section 10.9). I cover more exotic modules features such as parameterized interfaces and value sharing as well as some design alternatives briefly in a later chapter on possible extensions.

5.1 Higher Kinds

I have omitted any mention of functions on types from the first part of this dissertation in order to keep the discussion simple. Such functions are necessary to handle concepts like list and reference types: Statically-typed programming languages have not one list type, but rather a family of list types parametrized by the type of the elements their members may contain. Thus, an integer list may contain only integers and so on. Because of this parametrization, list types are most naturally introduced by making `list` a primitive function taking a type (the type of the list's elements) and returning a type (the resulting list type.) Thus, `list(int)` would denote a list of `int`'s and `list(bool)` would denote a list of `bool`'s.

Both types and functions from types to types are instances of a more general notion of *constructors*. Constructors are generalized types that may take any number of arguments. Types (e.g., `int`) are constructors that take zero arguments, while functions mapping a type to a type (e.g., `list`) are constructors taking one type argument. Constructors in my kernel system may also take multiple arguments through currying: For example, we might have a constructor `hash_table` that takes two arguments, namely the type of keys and the type of elements to use. Such a table with `string` keys and `int` elements would have type `hash_table(string)(int)`.

Constructors are distinct from terms, forming a separate level of my kernel system. In order to do type checking, it is necessary to distinguish among the different sorts of constructors based on how they may be applied. This distinguishing can be done by introducing a third level of *kinds*, which classify constructors in the same way that types classify terms. The simplest kind is **type**, which classifies completely applied constructors. Only such constructors may be used to classify terms; all other constructors must first be applied to suitable arguments before they can be used to classify terms. I shall refer to only constructors of kind **type** as types.

The remaining kinds, called higher kinds, are of the form $K_1 \Rightarrow K_2$, where K_1 and K_2 denote kinds. Such kinds classify constructors that map constructors of kind K_1 to constructors of kind K_2. Thus, `list` will have kind **type**$\Rightarrow$**type** and `hash_table` will have kind **type**$\Rightarrow$(**type**$\Rightarrow$**type**).

It is useful to allow programmers to define new constructors in terms of old ones. For example, a programmer might want to define dictionaries to be hash tables with `string` keys:

constr `dictionary =` $\lambda\alpha$::**type**. `hash_table(string)(`α`);`

This declaration introduces a new constructor, `dictionary`, of kind **type**$\Rightarrow$**type**. The intent is that `dictionary(`A`)` should equal `hash_table(string)(`A`)` for all types A.

My system supports this ability by allowing such constructors ($\lambda\alpha$::K. A), defining their equality in terms of substitution:

$$(\lambda\alpha{::}K.\,A)\,A' = [A'/\alpha]A \tag{BETA}$$

$$\lambda\alpha{::}K.\,A\,\alpha = A, \text{ where } \alpha \notin \mathrm{FV}(A) \tag{ETA}$$

These rules allow equations such as the following to be derived:

$(\lambda\alpha{::}\textbf{type}.$ `hash_table(string)(`α`))(int) = hash_table(string)(int)`

$\lambda\alpha{::}\textbf{type}.$ `list(`α`) = list`

In essence, what is going on here is that the constructor and kind levels in my system constitute a simply-typed lambda calculus with constructors taking the part of simply-typed terms and kinds taking the part of their types. Constructors are equal if when considered as simply-typed terms they evaluate via $\beta\eta$-reduction to the same "value". Like functions in the simply-typed lambda calculus, constructors can be higher-order, taking and returning non-type constructors as arguments; for example, $\lambda\alpha{::}\textbf{type}{\Rightarrow}\textbf{type}.\ \alpha$`(int)` is a valid constructor of kind $(\textbf{type}{\Rightarrow}\textbf{type}){\Rightarrow}\textbf{type}$. Similarly, partially applied constructors can also be manipulated. For example, the programmer could equivalently have declared the following:

constr `dictionary = hash_table(string);`

My constructor level is thus quite flexible. Actual programming languages tend to have more restricted constructor levels in order to simplify the programming model for the programmer. For example, it is common to allow constructors to take only types as arguments. However, these restrictions do not yield any real simplification in the system proofs: All of the significant complexities introduced by having a constructor level occur even if only user-defined functions from types to types are allowed. Accordingly, I saw no good reason to make any restrictions.

In a system with constructors, the natural thing to do is to allow translucent sums to contain constructor components instead of type components. I do this in my kernel system, allowing translucent sums like

```
Dict = module
           constr dictionary = hash_table(string);
           val    empty = ...;
           ...
       end
    <: interface
           constr dictionary :: type⇒type;
           val    empty : ∀α. dictionary(α);
           ...
       end
```

Here the programmer creates an abstract data *constructor*, a generalization of an ADT. `Dict.dictionary` is abstract constructor of kind **type**⇒**type**; the fact that the programmer used hash tables to implement dictionaries will be hidden outside this module because of the type coercion. If the coercion had been omitted, then `Dict` would have been given a transparent type containing the declaration **constr** `dictionary = hash_table(string)`, causing `Dict.dictionary` to be an abbreviation for `hash_table(string)`.

Note that we now need to specify the kind of constructors in opaque declarations but not in transparent ones. This fact is because the kind of a given constructor is easily inferred. Because types occur so often, it is useful to allow the **type** keyword to be used as syntactic sugar for a **constr** declaration with kind **type**. Aside from this change, there is no other change to translucent sums or their behavior. The system translucent sums are embedded in changes greatly, however, due to the enriched behavior of equality, which now works on constructors of all kinds not just types, and subtyping.

5.2 Previous Work

I described a much earlier version of this work in a previous paper with Robert Harper [20]. Aside from minor notational differences, there are only two changes between the system described in that paper and the one I have described up to this point. The first change is that I have moved the handling of reordering translucent-sum fields from the definition of term equivalence (translucent sums whose fields could be reordered without changing any dependencies were considered equivalent) to the subtyping rules. This change simplifies the equality relation and makes the basic kernel system easier to adapt to an implementation that prohibits reordering and allows dropping fields only from the end of a translucent sum; such implementations can be very efficient because subsumption under those conditions can be implemented as a no-op.

The second change concerns the definition of extended values. Originally, I had defined them by adding term variables and the selection operation to the grammar for call-by-value values. Because the call-by-value value definition is recursive, this resulted in extended values including additional terms over the current definition like **module val** `x=z;` **val** `y=`**module val** `a=3;` **end**`.a;` **end**. These terms did not provide any real benefit and complicate the explanation of how equality on types containing terms is handled.

5.3 Extended Interface Matching

I arrived at the kernel system by making a number of simplifications to the system I have described so far. I am now going to discuss each of these simplifications in turn, describing a series of increasingly simple systems, ending with the actual kernel system.

One simplification I made was to treat *extended interface matching* (reordering and dropping module fields using subsumption) as an extension, rather than building it into the kernel system directly. The intent is that this ability be provided by a separate preprocessing phase that elaborates an extended version of the kernel language with implicit extended interface matching into the kernel system by inserting coercion functions where necessary.

For example, if the input program at some point used implicit subsumption to convert the term M's type from **interface val** `x:int;` **val** `y:bool;` **end** to **interface val** `y:bool;` **val** `x:int;` **end**, then the elaborator would insert an application of the following function to M at that point:

```
functor (m:interface val x:int; val y:bool; end)
  module val y=m.y; val x=m.x; end
```

Note that no coercions would be required if only equality or forgetting had been used.

In essence, this is just treating where and how to insert field-manipulation coercions as a form of type inference to be performed by the elaborator. This decision allows the kernel system and its proofs to be simplified greatly. Note that showing soundness for the extended system reduces to just verifying that the output of the elaborator is always well typed in the kernel system, given a soundness proof for the kernel system. Another advantage of this choice is that it avoids hardwiring a particular set of subtyping rules into the kernel system; this is probably wise given that there is some uncertainly about what the best choice of subtyping rules is for handling component reordering (see Section 13.2) and the difficulty of changing the kernel-system proofs.

5.4 Unnamed Binary Sums

Without the ability to reorder or drop module fields using subsumption, there is no good reason in a kernel system to have labeled fields or n-ary (as opposed to binary) sums, so

I have eliminated those features as well (see Section 13.1 for how they might be added back as an extension). Thus, code like

```
M = module val x=3; val y=4; val z=y+1; end;

N = module constr T=int; end;

sum:N.t = M.x + M.y + M.z;
```

must first be translated to use only binary unnamed sums before it can be expressed in the kernel system. For example, the above code could be translated as

```
M = module
       val x=3;
       val module
              val y=4;
              val module val z=y+1; val 0; end;
          end;
    end;

N = module constr T=int; val 0; end;

sum:N.1 = M.1 + M.2.1 + M.2.2.1;
```

Here I am using the notation **module val** x=M_1; **val** M_2; **end** to denote a binary unnamed module with two value components, the first equal to M_1 and the second equal to M_2. These components can be selected only by number (i.e., $M.1$ or $M.2$). The syntax includes an internal name (x) for the first component that allows it to be referred to in M_2 (e.g., see the `y` and `z` fields above). This name is not in scope anywhere else and cannot be used to select the first component from outside the module. It thus acts like a local variable and can be α-varied as desired. Either or both components can be replaced by constructor components with the obvious syntax.

As a further simplification, I have removed the internal naming described above from simple-module expressions, yielding module creation constructs like **module val** M_1; **val** M_2; **end**. I did this because internal naming in simple-module expressions can be desugared into the use of a let statement, which in turn can be desugared to a function application:

module val x=M_1; **val** M_2; **end**

$\hookrightarrow$ **let** x=M_1 **in module val** x; **val** M_2; **end end**

$\hookrightarrow$ (**functor** (x:A) **module val** x; **val** M_2; **end**) M_1

where M_1 has type A. The case where the first component is a constructor is even simpler because the let statement in that case can be reduced using simple substitution:

$$\textbf{module constr}\ \alpha\texttt{=}A_1\texttt{;}\ \ \textbf{val}\ \ M_2\texttt{;}\ \ \textbf{end}$$

$$\hookrightarrow\ \textbf{let constr}\ \alpha\texttt{=}A_1\ \textbf{in}\ \textbf{module constr}\ \alpha\texttt{;}\ \textbf{val}\ M_2\texttt{;}\ \textbf{end}\ \textbf{end}$$

$$\hookrightarrow\ \textbf{module constr}\ A_1\texttt{;}\quad \textbf{val}\ [A_1/\alpha]M_2\texttt{;}\ \textbf{end}$$

Accordingly, internal naming is not essential in simple-module expressions. I have thus removed it in order to simplify the operational semantics of the kernel system by avoiding the need to include the following reduction step:

$$\textbf{module val}\ x\texttt{=}V_1\texttt{;}\ \ \textbf{val}\ \ M_2\texttt{;}\ \ \textbf{end}$$

$$\hookrightarrow\ \textbf{module val}\ V_1\texttt{;}\quad \textbf{val}\ [V_1/x]M_2\texttt{;}\ \textbf{end}$$

Note that internal naming is still present and needed in interfaces because not all dependencies can be broken. Consider, for example, the following interfaces for unnamed binary sums:

```
interface constr T::type; val list(T); end

interface
  val    M:interface constr T::type; constr ::type; end;
  constr FUNCTOR (x:M.T):M.U;
end
```

5.5 Reified Constructors

As a further simplification, I have factored the unlabeled binary translucent sums of the previous section into two simpler constructs, a standard dependent sum and a new construct called a *reified constructor*. A standard dependent sum (**module** M_1; M_2; **end** : **interface** x:A_1; A_2; **end**) is a binary unlabeled module that contains only value components. Only the fact that the type of its second component may depend on its first component distinguishes it from an ordinary pair.

In order to encode translucent sums using standard dependent sums, it is necessary to have some way of encoding constructor components as value components. Reified constructors provide a way to do this encoding. They allow taking a constructor A and reifying it into a term, $<A>$. The resulting reified-constructor term is an ordinary value, which can be manipulated in any of the usual ways. The constructor contained in a reified-constructor extended value ν can be *extracted* by writing $\nu!$. Constructor extraction (⇔!) is limited to extended values because of the extended-value restriction.

Reified constructors can be given either opaque ($<K>$) or transparent ($<=A::K>$) types; K here specifies the kind of the constructor the reified-constructor term contains. Similarly to the translucent-sum case, $\nu!$ will be equal to A if ν has the transparent type $<=A::K>$. Essentially, a reified constructor is just a degenerate form of translucent sum that contains exactly one unlabeled constructor field. The behavior and type rules of standard dependent sums and reified constructors are what you would expect from applying the original translucent-sum rules to translucent sums restricted either to have only two unlabeled value components or one unlabeled constructor component respectively.

Any unlabeled binary translucent sum can be easily encoded into one standard dependent sum and zero to two reified constructors where each constructor field in the original translucent sum is represented using a reified constructor. Some representative examples follow:

$$\begin{array}{lllll}
 & \mathbf{module} & \mathbf{val}\ M_1; & \mathbf{val}\ M_2; & \mathbf{end}\ : \\
 & \mathbf{interface} & \mathbf{val}\ x{:}A_1; & \mathbf{val}\ A_2; & \mathbf{end} \\
\hookrightarrow & \mathbf{module} & M_1; & M_2; & \mathbf{end}\ : \\
 & \mathbf{interface} & x{:}A_1; & A_2; & \mathbf{end}
\end{array}$$

$$\begin{array}{lllll}
 & \mathbf{module} & \mathbf{val}\ M_1; & \mathbf{constr}\ A_2; & \mathbf{end}\ : \\
 & \mathbf{interface} & \mathbf{val}\ x{:}A_1; & \mathbf{constr}\ ::K_2; & \mathbf{end} \\
\hookrightarrow & \mathbf{module} & M_1; & <A_2>; & \mathbf{end}\ : \\
 & \mathbf{interface} & x{:}A_1; & <K_2>; & \mathbf{end}
\end{array}$$

$$\begin{array}{lllll}
 & \mathbf{module} & \mathbf{constr}\ A_1; & \mathbf{val}\ M_2; & \mathbf{end}\ : \\
 & \mathbf{interface} & \mathbf{constr}\ \alpha\texttt{=}A_1; & \mathbf{val}\ A_2; & \mathbf{end} \\
\hookrightarrow & \mathbf{module} & <A_1>; & M_2; & \mathbf{end}\ : \\
 & \mathbf{interface} & x{:}<=A_1::K_1>; & [x!/\alpha]A_2; & \mathbf{end}
\end{array}$$

$$\begin{array}{lllll}
 & \mathbf{module} & \mathbf{constr}\ A_1; & \mathbf{constr}\ A_2; & \mathbf{end}\ : \\
 & \mathbf{interface} & \mathbf{constr}\ \alpha::K_1; & \mathbf{constr}\ A_2; & \mathbf{end} \\
\hookrightarrow & \mathbf{module} & <A_1>; & <A_2>; & \mathbf{end}\ : \\
 & \mathbf{interface} & x{:}<K_1>; & [x!/\alpha]<=A_2::K_2>; & \mathbf{end}
\end{array}$$

Note that we no longer need to allow transparent declarations (**constr** α**=**A) in assignments because they are not needed to type check dependent sums or reified constructors and because their presence in assignments can be encoded away via a similar encoding.

This factoring simplifies the system in two main ways. First, it avoids the need to discriminate on the sort of fields a translucent sum has. If the field sorts matter, then there are four distinct cases that may need to be considered under the original system; if the transparency/opacity of a constructor component matters as well, then the number rises to nine cases. (E.g., the need to give four different sample encodings above.) In the factored system, only one case of sums and possibly two cases of reified constructors ever need to be considered.

Second, it eliminates the redundancy in the original system of having two separate ways of specifying a constructor abbreviation: directly via a constructor field (e.g., **constr** α=A in an assignment ,) and indirectly via a constructor field of a translucent sum. Because of this redundancy, for example, the original system I described needed two rules to handle equality for constructor abbreviations (ABBREV and ABBREV$'$) whereas the kernel system needs only one. In the kernel system, constructor abbreviations are specified only indirectly via a reified constructor. The original formulation is likely to be preferred for an actual programming language because it probably produces a slightly more efficient implementation in practice (see Section 13.1).

5.6 Rules and Restrictions

As another simplification, I changed where the EVALUE rules can be applied. Originally, they could be used at any point in a proof, so long as the term being typed was an extended value. However, it is not hard to see that any such proof can be normalized so that the EVALUE rules are only used immediately after a term variable has been introduced on that variable and its (sub)-components. (The EVALUE rules are not needed to give fully-defined transparent minimal types to the other extended values.) Because such proofs are easier to reason about, I have restricted the application of the EVALUE rules to places (term variables and their (sub)-components) so that only such normalized proofs can be constructed. This restriction has no effect on what terms can be given what types.

I have also further restricted what term expressions are allowed in constructors so that only places ($x.i_1 \cdots .i_n$) may occur in constructors. In the system I described in the first part of this dissertation, arbitrary extended values could appear in constructors. However, in that system any valid constructor containing extended values could be replaced with an equal valid constructor that contains only places. Because the equality of two constructors that contain only places need never involve constructors that contain terms other than places (this can be shown via a confluence argument), the ability to have constructors containing arbitrary extended values was really useful only as a way of explaining how substitution of values for term variables in constructors works.

In the kernel system, value substitution is instead handled directly via a special notion of substitution (see Section 11.6) that does any selections needed to reduce values to places at substitution time. For example, the following holds using this notion of substitution:

`[module <`A_1`>; <`A_2`>; end/x]x.2! = `A_2

This restriction to places allows a more specialized method (see Section 6.4) to be used to determine the identity and kind of a constructor extraction, breaking the dependency in the first part's system on the subtyping and well-formed term judgments by the constructor validity and equality judgments. (These dependencies are caused by the V-DOT and ABBREV′ rules, which require typing the term ν — possibly using subtyping — in order to decide if $\nu.\alpha$ is valid or what it is equal to.) Removing these dependencies greatly simplifies the proofs.

I have also introduced a new *self* function similar to Leroy's strengthening operation (see Sections 11.1 and 14.3 for details), which captures the effect on a place's type of applying a series of EVALUE rules in a row. More precisely, if starting from the fact that $\Gamma \vdash x.i_1 \cdots .i_n : A$ we can derive the type A' for $x.i_1 \cdots .i_n$ using just the EVALUE and I-SELECT rules, then we can get a type equal to A' under Γ by applying the function [self=$x.i_1 \cdots .i_n$] to a type equal to A under Γ. The introduction of self allowed me to further normalize the proofs by replacing the EVALUE rules by a single application of the self function when term variables are introduced.

5.7 Notation

Because of the great volume of the proofs, I shall use a more concise, mathematical, and traditional notation to describe the actual kernel system. Figure 5.1 contains a translation key from the notation I have used before this point into the corresponding kernel-system notation. In addition to the notation shown in this figure, the kernel system has two primitive constructors (**rec** and **ref**) and five primitive functions (**roll**, **unroll**, **new**, **get**, and **set**). These provide the functionality of recursive types and references (mutable cells) and are described in the next section.

5.8 System Summary

In this section I provide a mostly self-contained quick overview of my kernel system. Complete details will be provided in later chapters as I consider each part of the system in turn.

My kernel system is based on Girard's F_ω [15] in much the same way that many systems are based on the second-order lambda calculus (F_2). That is to say, my system can

Dependent Sums:	**module** M_1; M_2; **end**	(M_1, M_2)
	interface $x : A_1$; A_2; **end**	$\Sigma x{:}A_1.\, A_2$
	$M.i$	$M.i$
Reified Constructors:	$<A>$	$<A>$
	$<K>$	$<K>$
	$<=A{::}K>$	$<=A{::}K>$
	$x.i_1 \cdots .i_n!$	$x.i_1 \cdots .i_n!$
Functors:	**functor** $(x : A)$ M	$\lambda x{:}A.\, M$
	FUNCTOR $(x : A)$ A'	$\Pi x{:}A.\, A'$
	M_1 M_2	$M_1\, M_2$
Type Coercions:	$M{<:}A$	$M{<:}A$
Constructors:	$\lambda\alpha{::}K.\, A$	$\lambda\alpha{::}K.\, A$
	$A_1(A_2)$	$A_1\, A_2$
Kinds:	**type**	Ω
	$K_1{\Rightarrow}K_2$	$K_1{\Rightarrow}K_2$
Declarations:	**val** $x : A$	$x{:}A$
	constr $\alpha : : K$	$\alpha{::}K$
Assignments:	$D_1; \ldots; D_n$	$\bullet, D_1, \ldots, D_n$

Figure 5.1: The translation into kernel-system notation

be (roughly) thought of as being obtained from F_ω by adding more powerful constructs (dependent functions, dependent sums, reified constructors, references, and recursive types) and a notion of subtyping and then removing the old constructs (quantification[1] ($\forall$), weak sums ($\exists$), and non-dependent functions ($\rightarrow$)) superseded by the new ones. Subtyping interacts with the rest of the calculus via implicit subsumption. Bounded quantification is not supported.

Like F_ω, my system is divided into three levels: terms, (type) constructors, and kinds. Kinds classify constructors, and a subset of constructors, called types, having kind Ω classify terms. The kind level is necessary because the constructor level contains functions on constructors. Example: the constructor $\lambda\alpha{::}\Omega.\,\mathbf{ref}\,\alpha$ has kind $\Omega{\Rightarrow}\Omega$ and when applied to type int yields **ref** int. There is no separate module system level; all terms including dependent sums, dependent functions, and reified constructors are first-class.

5.8.1 Module Constructs

Grouping in my system is handled by (binary unlabeled) dependent sums. Dependent-sum terms are written $(M_1,\ M_2)$ with corresponding type $\Sigma x{:}A.\,A'$. As the form of their type indicates, the type of their second component may depend on the value of their first component. Ordinary pairs are a degenerate form of dependent sums where there is no dependency. The elimination forms are $M.1$ for the first component of M and $M.2$ for the second component of M.

Functional abstraction is handled in my system by dependent functions, denoted by $\lambda x{:}A.\,M$ with corresponding type $\Pi x{:}A.\,A'$. The result type of an application $(M_1\,M_2)$ may depend on the value of the argument (M_2). Ordinary functions are a degenerate form of dependent functions where there is no dependency.

The ability to include constructor components in a module with optional identity information is handled by reified constructors. Reified constructors allow packing up a constructor A of kind K into a term, written $<\!A\!>$. This term has both a transparent type, $<\!=\!A{::}K\!>$, with information on the identity of its constructor component, and an opaque type, $<\!K\!>$, with no such information. The opaque type is a supertype of the transparent one, allowing the component information to be forgotten later.

Because terms in constructors are limited to places, the elimination form for reified constructors is syntactically limited to places. If $x.i_1.i_2\cdots.i_n$ $(n \geq 0)$ denotes a reified constructor, then $x.i_1.i_2\cdots.i_n!$ denotes the constructor it contains. If $x.i_1.i_2\cdots.i_n$ has type $<\!=\!A{::}K\!>$ under assignment , then the equation $x.i_1.i_2\cdots.i_n! = A$ will hold under , .

[1]Quantification is derivable from dependent functions and reified constructors. See Section 5.8.1 for details.

Quantification is a derived form in my system, resulting from a combination of reified constructors and dependent functions. The quantified term $\Lambda\alpha{::}K.M$, which denotes the term M parametrized by a constructor of kind K named α, can be regarded as standing for $\lambda x{:}{<}K{>}.\,[x!/\alpha]M$, where x is not free in M, and the instantiation $M\ [A]$, which denotes instantiating the quantified term M with the constructor A, can be regarded as standing for $M\,{<}A{>}$.

Aside from the exceptions already mentioned (forgetting component information, constructor abbreviations), the equality and subtyping rules are standard. The programmer can use an explicit coercion (written $M{<}{:}A$) where desired to force a term M to be coerced to a supertype A. Coercions in the kernel system have no semantic effect; their only effect is to discard type information. Coercions may have semantic effects in extensions however. Also present is the machinery (a EVALUE-like rule, restrictions in the elimination rules for dependent sums and products) needed to properly propagate information about constructor components solely through the type level.[2]

5.8.2 References and Recursive Types

SML-like references are provided by three built-in primitives: **new**, **get**, and **set**. If M is a term that denotes a value of type A then $\mathbf{new}\,{<}A{>}\,M$ returns a new reference with initial value M. The resulting reference has type $\mathbf{ref}\,A$. Given a term M that denotes a reference containing values of type A, $\mathbf{get}\,{<}A{>}\,M$ returns the current value of that reference. Finally, given a term M_1 that denotes a reference containing values of type A and a term M_2 denoting a value of type A, $\mathbf{set}\,{<}A{>}\,M_1\,M_2$ sets the reference's value to be M_2 then returns that value.

Recursive types are implemented by means of an isomorphism between $\mathbf{rec}\,A$ and $A\,(\mathbf{rec}\,A)$ mediated by two primitives **roll** and **unroll** (here A is of kind $\Omega{\Rightarrow}\Omega$). $\mathbf{roll}\,{<}A{>}$ maps terms of type $A\,(\mathbf{rec}\,A)$ to terms of type $\mathbf{rec}\,A$ and $\mathbf{unroll}\,{<}A{>}$ does the opposite. Semantically, for properly typed V_1 and V_2's, $\mathbf{unroll}{<}A{>}\,(\mathbf{roll}{<}A{>}\,V_1) = V_1$ and $\mathbf{roll}{<}A{>}\,(\mathbf{unroll}{<}A{>}\,V_2) = V_2$. As an example of the use of recursive types, I have sketched in Figure 5.2 an implementation of integer lists written in SML-like pseudo-code (rather than kernel-system notation) using some simple extensions to the kernel system.

In the example, the term {} denotes the only value of the type `unit` and the type $A_1 + A_2$ denotes a disjoint union type; the disjoint union's values are `inLeft(`V_1`)` and `inRight(`V_2`)` where V_1 is of type A_1 and V_2 is of type A_2. The disjoint union destructors are `asLeft` and `asRight`; they raise the exception `fail` if their argument is of the wrong sort. The function `isLeft` checks to see if its argument is of the left sort. I have omitted "type information", such as applications to <`intIter`>, that type inference might supply. (My system does not have a unit type, disjoint union's, exceptions, or

[2]The discussion of this machinery, how it works, and why it is needed, can be found in Chapter 4.

```
(*
 * Set things up so that intList ≅ unit + (int, intList)
 * via roll and unroll:
 *)
constr intIter = λα::Ω. unit + (int, α);
type   intList = rec intIter;

val  nil = roll (inLeft {}) ;
fun  cons(x:int, y:intList) = roll (inRight (x, y));

fun  hd(x:intList)    = (asRight(unroll x)).1;
fun  tl(x:intList)    = (asRight(unroll x)).2;

fun  isNil(x:intList) = isLeft(unroll x);
```

Figure 5.2: Implementing integer lists with recursive types

extensive type inference; I am using them here only for illustrative purposes.)

5.8.3 Semantics

Evaluation proceeds in a deterministic manner from left to right in a call-by-value manner. Evaluation does not proceed under quantifications because quantification translates into a lambda abstraction. My system differs in this regard from SML which does evaluate under quantifications. See [19] for a discussion of the differences between these two interpretations of quantifications and why this choice seems to be preferable. In the terminology of that paper, my system implements the standard call-by-value semantics for F_ω extended with the obvious implementations of references and recursive types. Note that non-termination is a possible result of evaluation because of the inclusion of recursive types. (E.g., recursive types can be used to implement a fixed point operator.)

5.9 On the Nature of the Proofs

Even after all the simplifications I have implemented, my kernel system is still a large and complicated system when compared with the sort of systems that are usually analyzed formally. For example, the system has 48 type rules using 8 auxiliary functions, 10 ways of building constructors, 13 ways of building terms, and 8 evaluation rules. There are also a number of dependencies in the system not possessed by simpler systems: constructors can depend on term variables, the validity of constructors and the equality of constructors judgments are mutually recursive, and the equality of two constructors depends on the assignment under which they are compared.

These dependencies greatly increase the difficulty level of the needed proofs. For example, they prevent using any of the standard methods for establishing normalization or confluence of constructors. Novel methods or extensions to existing methods must be invented to handle them.

Mutually dependent definitions also reduce the modularity of the proofs. It is not possible, for example, to separate the results about the validity of constructors from those about the equality of constructors. Indeed, the intertwined proofs of those results take up four full chapters! The choices of proof methods for the various parts are also highly interdependent: Proof methods in order to work well often require that the type system be structured in certain ways; unfortunately, these requirements often conflict, preventing proof methods for different parts of the system from being chosen independently.

These facts make the process of designing and proving correct the kernel system resemble that of solving an NP-complete problem: There are many choice points and the complexity and sheer size of the problem prevent effective lookahead, resulting in extensive guess work and backtracking until a solution is found. Many times I discovered only after months of work that a particular proof method was unusable and had to be discarded because of a subtle interaction with another part of the system.

Like what often happens when solving real NP problems, time limits have prevented me from finding an optimal solution. There was simply no time to try each choice to see which is best; many of my choices are accordingly just the first choice I tried that worked. Better choices almost certainly exist. I have briefly summarized at the major choice points which alternatives I considered, why I tried the ones I did, and why many of them failed to work. Space limitations and the complexity of the interactions between many choices prevent me from giving more details. I have also included speculations on which choices seem better based on hindsight. It is my hope that these will benefit future designers of systems like this one.

5.10 Organization

I have split up the description of my system and the accompanying proofs into four major sections: constructor validity and equality, subtyping, soundness, and type checking. To improve the presentation, I delay introducing parts of the system until they are needed. For example, terms are not discussed until the section on soundness. For the reader's convenience, Appendix A contains a complete copy of all the system definitions together in one place. Briefly, the material covered in each of the chapters devoted to the kernel system is as follows.

Chapters 6 to 9 cover constructor validity and equality as a unit. Chapter 6 introduces the kind and constructor levels of the system and gives the rules for the valid constructor and equality judgments. It also introduces an alternative formulation called the tagged system where term variables in types are tagged with their assigned type. These tags allow the equality of tagged constructors to be decided independently of the assignment they are being compared under. Chapters 7 and 8 prove the needed results about the more tractable tagged system which are then transferred back to the original system in Chapter 9.

In Chapter 7, I use fairly standard methods to introduce a rewriting relation on constructors for the tagged system and show that it is confluent, that it preserves typing, and that it implements equality.[3] Because rewriting also preserves the outer shape of a constructor (roughly, is it a function type?, a sum type?, etc.), I am also able to establish results about what sort of outer shapes two equal constructors can have as well as show that equal constructors of the same outer shape must have equal subcomponents under the appropriate assignments. One consequence, for example, is that function types can never be equal to sum types.

In Chapter 8, I give an algorithm for deciding if a constructor in the tagged system is valid. In the process of validity checking, the algorithm reduces the constructor using the previously introduced rewrite relation. If the constructor is found to be valid, the algorithm returns a normal form for that constructor. Using this algorithm, I show that all the judgments of the tagged system are decidable and that normal forms for valid constructors are computable.

In Chapter 9, I show how to translate judgment derivations back and forth between the two systems using notions of *tag removal* and *stamping*. I then translate the results from the previous two chapters on the tagged system to the original system. This translation shows, for example, that the original valid constructor and equality judgments are decidable. The translation of the normal form results is somewhat tricky and is handled by what amounts to an abstract data type: The definition of the untagged normal form property uses the tagged system; however, I provide sufficient theorems so that later uses of the property need not use the the tagged system results directly.

Chapter 10 introduces the subtyping relation. Much of the chapter is devoted to proving that judgments may be weakened by replacing a type in the assignment with a subtype of the original type. I also establish results about the shape and sub-components of subtypes. Using these results, I prove correct a semi-decision procedure for subtyping. I show that one cannot do better by proving that subtyping is undecidable.

Chapter 11 introduces the term level and lays out the kernel system's semantics. The weakening by a subtype result from the previous chapter is extended to the well-typed

[3]By implements equality, I mean that the reflexive, symmetric, and transitive closure of the rewriting relation is the same as the equality relation modulo the validity of the argument constructors.

term judgment with some difficulty due to the self function . Using this result, I show that the system is sound — evaluation is deterministic, does not become stuck, and preserves typing.

Finally, Chapter 12 discusses type checking. To make the problem more tractable, I remove some of the built-in type inference in the system. In particular, some coercions must now be indicated explicitly by the programmer instead of being automatically inferred. I show that the restricted well-typed term judgment is semi-decidable (decidable modulo subtyping) by proving correct a decision procedure.

Chapter 6

Types: Setup

Beginning in this chapter, and continuing for the next three chapters, I discuss the kind and constructor levels of my kernel system, including the notions of constructor validity and constructor equality, but omitting the notion of subtyping which I discuss in Chapter 10. In particular, I establish the major results about these notions, namely decidability and the results about how constructors behave under equality. I also establish numerous minor results such as strengthening that are needed for later proofs.

6.1 Kinds and Constructors

Definition 6.1.1 (Syntax for the constructor and kind levels)

Kinds	K	$::=$	$\Omega \mid K{\Rightarrow}K'$
Constructors	A	$::=$	$\alpha \mid \Pi x{:}A.\,A' \mid \Sigma x{:}A.\,A' \mid \lambda\alpha{::}K.\,A \mid A\,A' \mid$
			$<\!K\!> \mid <\!=\!A{::}K\!> \mid x\rho! \mid$ **rec** $\mid$ **ref**
Paths	ρ	$::=$	$\epsilon \mid \rho.1 \mid \rho.2$
Assignments	,	$::=$	$\bullet \mid , , D$
Declarations	D	$::=$	$\alpha{::}K \mid x{:}A$

Here, the metavariable α ranges over *constructor variables* and the metavariable x ranges over *term variables*. The kind Ω denotes the kind of constructors that classify terms (i.e., types). The empty path is denoted by ϵ and the empty assignment by $\bullet$. For convenience, I shall sometimes omit them in non-empty paths and assignments, writing, for example, .1 for ϵ.1 and $\alpha{::}K$ for $\bullet, \alpha{::}K$. I shall refer to $x\rho$'s as *places* and to $x\rho!$'s as *constructor extractions.*

Objects of the kernel system are considered equivalent if they differ only by α-conversion; the only exception is that assignments with no redeclared variables are never considered equivalent to assignments with redeclared variables. This exception allows the valid assignment judgment to require that valid assignments never redeclare variables, simplifying the system. Scoping is as expected and we have the obvious notions of free constructor variables (FCV($\Leftrightarrow$)), free term variables (FTV($\Leftrightarrow$)), and constructor substitution ($[A/\alpha]\Leftrightarrow$) on constructors, declarations, and assignments:

Definition 6.1.2 (Free constructor variables)

$$
\begin{array}{lcl}
FCV(\alpha) & = & \{\alpha\} \\
FCV(\lambda\alpha{::}K.\,A) & = & FCV(A) \Leftrightarrow \{\alpha\} \\
FCV(\Pi x{:}A_1.\,A_2) & = & FCV(A_1) \cup FCV(A_2) \\
FCV(\Sigma x{:}A_1.\,A_2) & = & FCV(A_1) \cup FCV(A_2) \\
FCV(A_1\,A_2) & = & FCV(A_1) \cup FCV(A_2) \\
FCV(<=A{::}K>) & = & FCV(A) \\
FCV(<K>) & = & \emptyset \\
FCV(x\rho!) & = & \emptyset \\
FCV(\mathbf{rec}) & = & \emptyset \\
FCV(\mathbf{ref}) & = & \emptyset \\
 & & \\
FCV(\alpha{::}K) & = & \emptyset \\
FCV(x{:}A) & = & FCV(A) \\
 & & \\
FCV(\bullet) & = & \emptyset \\
FCV(,\,,D) & = & FCV(,\,) \cup FCV(D)
\end{array}
$$

Definition 6.1.3 (Free term variables)

$$\begin{array}{rcl}
FTV(x\rho!) & = & \{x\} \\
FTV(\Pi x{:}A_1.\, A_2) & = & FTV(A_1) \cup (FTV(A_2) \Leftrightarrow \{x\}) \\
FTV(\Sigma x{:}A_1.\, A_2) & = & FTV(A_1) \cup (FTV(A_2) \Leftrightarrow \{x\}) \\
FTV(\lambda\alpha{::}K.\, A) & = & FTV(A) \\
FTV(A_1\, A_2) & = & FTV(A_1) \cup FTV(A_2) \\
FTV(<=A{::}K>) & = & FTV(A) \\
FTV(\alpha) & = & \emptyset \\
FTV(<K>) & = & \emptyset \\
FTV(\mathbf{rec}) & = & \emptyset \\
FTV(\mathbf{ref}) & = & \emptyset \\
\\
FTV(\alpha{::}K) & = & \emptyset \\
FTV(x{:}A) & = & FTV(A) \\
\\
FTV(\bullet) & = & \emptyset \\
FTV(, , D) & = & FTV(,) \cup FTV(D)
\end{array}$$

Definition 6.1.4 (Constructor substitution)

$$\begin{array}{rcll}
[A/\alpha]\alpha & = & A & \\
[A/\alpha]\alpha' & = & \alpha' & (\alpha \neq \alpha') \\
[A/\alpha]\lambda\alpha'{::}K.\, A' & = & \lambda\alpha'{::}K.\, [A/\alpha]A' & (\alpha' \neq \alpha,\ \alpha' \notin FCV(A)) \\
[A/\alpha]\Pi x{:}A_1.\, A_2 & = & \Pi x{:}[A/\alpha]A_1.\, [A/\alpha]A_2 & (x \notin FTV(A)) \\
[A/\alpha]\Sigma x{:}A_1.\, A_2 & = & \Sigma x{:}[A/\alpha]A_1.\, [A/\alpha]A_2 & (x \notin FTV(A)) \\
[A/\alpha](A_1\, A_2) & = & [A/\alpha]A_1\, [A/\alpha]A_2 & \\
[A/\alpha]<=A'{::}K> & = & <=[A/\alpha]A'{::}K> & \\
[A/\alpha]<K> & = & <K> & \\
[A/\alpha]x\rho! & = & x\rho! & \\
[A/\alpha]\mathbf{rec} & = & \mathbf{rec} & \\
[A/\alpha]\mathbf{ref} & = & \mathbf{ref} & \\
\\
[A/\alpha](\alpha'{::}K) & = & \alpha'{::}K & \\
[A/\alpha](x{:}A') & = & x{:}[A/\alpha]A' & \\
\\
[A/\alpha]\bullet & = & \bullet & \\
[A/\alpha](, , D) & = & ([A/\alpha],), [A/\alpha]D &
\end{array}$$

Because paths and assignments are lists, it is useful from time to time to have a notation for appending them. I shall use $\rho_1\rho_2$ to denote ρ_2 appended to ρ_1 and $,_1; ,_2$ to denote $,_2$ appended to $,_1$. Assignments can also be regarded usefully as partial functions:

Definition 6.1.5 (Assignment regarded as a partial function)

$$\begin{array}{rcll}
dom(\bullet) & = & \emptyset & \\
dom(,\ , x{:}A) & = & dom(,\ ) \cup \{x\} & \\
dom(,\ , \alpha{::}K) & = & dom(,\ ) \cup \{\alpha\} & \\
 & & & \\
(,_1; x{:}A; ,_2)(x) & = & A & \qquad\qquad (x \notin dom(,_1))
\end{array}$$

With the exception of subtyping, which shall be discussed later, there are four judgments at the kind and constructor levels:

Definition 6.1.6 (Judgments)

$$\begin{array}{rl}
\vdash ,\ \ valid & \quad valid\ assignment \\
 & \\
,\ \vdash A :: K & \quad valid\ constructor \\
,\ \vdash A = A' :: K & \quad equal\ constructors \\
 & \\
,\ \vdash x\rho \Rightarrow A & \quad place\ lookup
\end{array}$$

All of these judgments are defined mutually recursively. For example, the valid assignment judgment checks that each constructor appearing in a valid assignment is valid under the immediately preceding part of that assignment. Place lookup is an auxiliary judgment used by the valid and equal constructor judgments.

The kind of the constructor $x\rho!$ and what it is equal to depend on the type of the *term* $x\rho$. For example, if $x\rho$ has type $<\!=\!A{::}K\!>$ then $x\rho$ denotes a reified constructor containing a constructor of kind K that is known to be equal to A so $x\rho!$ has kind K and is equal to A. The place lookup judgment is used to determine the type of terms of this form. If $,\ \vdash x\rho \Rightarrow A$ then $x\rho$ has type A under assignment , .

Rather than being defined directly in terms of the well-formed term judgment, which assigns types to terms, the place lookup judgment is defined using a highly specialized copy of the machinery needed to determine a term's type. This copying and specialization allows the valid and equal constructor judgments to not depend on the subtyping or well-formed term judgments, greatly simplifying the system.

I shall present the details of these four judgments shortly, but first I need to discuss my overall proof strategy so the organization of the rules will make sense.

6.2 Proof Strategy

The key result that needs to be established about the valid and equal constructor judgments is the existence of a confluent rewriting relation that implements equality and under which all valid constructors have normal forms. This result allows the equality of two valid constructors to be easily tested by simply normalizing both constructors then testing them to see if they are the same modulo α-conversion. More generally, the same procedure can also be used to determine constructively if a given valid constructor is equal to any valid constructor of a certain shape. This ability is needed in order to show constructor validity checking decidable.

Unfortunately, because the equality of two constructors in my system depends on the assignment under which they are compared (e.g., under $x{:}{<}{=}A{::}\Omega{>}$, $x! = A$), the standard methods for establishing confluence cannot be used. After extensive searching, I have found only two approaches that seem promising. Both are based on the well-known idea of showing confluence via a rewriting relation which has the diamond property.

A rewriting relation has the diamond property if for any A that rewrites to A_1 and to A_2, each in a single step, we can find an A' such that A_1 and A_2 both rewrite to A' in a single step. Confluence follows immediately from this property via a simple "fill-in-the-grid" proof (see Section 7.2 for details). In order to use this proof method effectively, the rewriting relation must be *simple*. By simple, I mean roughly that a single step of the rewriting relation must do only a small amount of work on each sub-constructor. If the rewriting relation is not simple then proving the diamond property is no easier than proving confluence directly.

The simple property is not easy to arrange in the presence of reified constructors. In particular, the obvious rule for rewriting $x!$ is as follows. (I am ignoring paths throughout this section to simplify the discussion.)

$$\frac{,\,(x) \rightarrow^*_{,} {<}{=}A{::}K{>}}{x! \rightarrow_{,} A} \qquad \text{(OBVIOUS)}$$

Note that this rule can do an arbitrary amount of work ($\rightarrow^*$ denotes the transitive closure of $\rightarrow$) in a single step and is thus not simple.

Each of my two possible approaches handles the problem of how to obtain a simple rewriting relation in a different way. In the *assignment* approach, assignments are rewritten in parallel with constructors. The rule for rewriting $x!$ can then deal with only the last step of rewriting the type of x:

$$\frac{,\,(x) \rightarrow_{,} {<}{=}A{::}K{>}}{x! \rightarrow_{,} A} \qquad \text{(ASSIGN)}$$

The main rule for rewriting assignments is as follows:

$$\frac{, \rightarrow ,' \qquad A \rightarrow_{,'} A'}{,\, ,\, x{:}A \rightarrow ,',\, x{:}A'} \qquad \text{(STAGED)}$$

Note that A is rewritten using $,'$ rather than $,$; this is necessary to obtain the diamond property. Under this approach A and A' will be equal under $,$ iff $,, x{:}A$ and $,, x{:}A'$ ($x \notin \text{dom}(,)$) have a common reduct.

The *Tagging* approach, on the other hand, introduces a separate tagged system where term variables are tagged with their type (written x_A). This allows x's type to be rewritten in a series of steps before $x!$ is reduced:

$$\frac{A_1 \to A_2}{x_{A_1}! \to x_{A_2}!} \qquad \text{(TAG)}$$

$$\frac{A \to \,<\!=\!A'{::}K\!>}{x_A! \to A'} \qquad \text{(REDUCE)}$$

The type rules of the tagged system require that if x_A occurs in a constructor valid under $,$ then $,(x)$ is equal to A under $,$. This requirement ensures that the tags are consistent with the assignment. In essence, where the assignment approach reduces x's type in place (in the assignment), the tagging approach reduces a local copy of x's type.

The tagging approach has the advantage that its rewriting relation does not depend on the assignment. This fact greatly simplifies its confluence and normalization proofs. However, once proved, the results on the tagged system must be carefully transfered back to the original untagged system because tagging is incompatible with subtyping. (Subtyping alters assignments, replacing types with subtypes or supertypes; there is no good way to maintain the consistency of the tags with the assignment in this case.) This transfer can be accomplished by transforming derivations between the two systems using tag removal and stamping (replacing x by $x_{,(x)}$ where $,$ is the current assignment).

It is not at all obvious which of these two approaches is better. The tagging approach seems to have simpler proofs but is more verbose (two systems instead of one) and requires extra work to transfer the results. Given the information I had at the time, I chose to go with the tagging approach. In hindsight, it is not clear that this was the best choice. I discuss further, using the benefit of hindsight, this and other decisions in Section 9.7.

Summarizing, my basic proof strategy for Chapters 6 to 9 is to do the following:

- Describe the original (untagged) system
- Describe the corresponding tagged system
- Describe a rewriting relation that implements tagged equality
- Prove the rewriting relation confluent via the diamond property
- Show normalization and the other desired results for the tagged system
- Transfer the results back to the original system

Note that in order to make the transfer and equality implementation proofs easy, I have designed the two systems and the rewrite relation to mirror each other as much as possible.

6.3 The Type Rules

The rules for the (untagged) judgments we are considering are given below. Aside from the rules dealing with reified constructors (C-OPAQ, C-TRANS, C-EXT-O, C-EXT-T, E-TRANS, and E-ABBREV) and place lookup (P-INIT and P-MOVE), they are standard. I shall defer discussion of the place lookup rules and the defining of the selection function ($\mathcal{S}(A, x\rho, \rho')$) until the next section.

Definition 6.3.1 (Assignment Formation Rules)

$$\vdash \bullet \ \mathit{valid} \qquad \text{(EMPTY)}$$

$$\frac{\vdash , \ \mathit{valid} \qquad \alpha \notin \mathit{dom}(,)}{\vdash , , \alpha{::}K \ \mathit{valid}} \qquad \text{(DECL-C)}$$

$$\frac{, \vdash A :: \Omega \qquad x \notin \mathit{dom}(,)}{\vdash , , x{:}A \ \mathit{valid}} \qquad \text{(DECL-T)}$$

Definition 6.3.2 (Constructor Formation Rules)

$$\frac{\vdash , \ \mathit{valid} \qquad \alpha{::}K \in ,}{, \vdash \alpha :: K} \qquad \text{(C-VAR)}$$

$$\frac{, , x{:}A \vdash A' :: \Omega}{, \vdash \Pi x{:}A.\, A' :: \Omega} \qquad \text{(C-DFUN)}$$

$$\frac{, , x{:}A \vdash A' :: \Omega}{, \vdash \Sigma x{:}A.\, A' :: \Omega} \qquad \text{(C-DSUM)}$$

$$\frac{, , \alpha{::}K \vdash A :: K'}{, \vdash \lambda\alpha{::}K.\, A :: K{\Rightarrow}K'} \qquad \text{(C-LAM)}$$

$$\frac{, \vdash A_1 :: K_2{\Rightarrow}K \qquad , \vdash A_2 :: K_2}{, \vdash A_1\, A_2 :: K} \qquad \text{(C-APP)}$$

$$\frac{\vdash , \ \mathit{valid}}{, \vdash {<}K{>} :: \Omega} \qquad \text{(C-OPAQ)}$$

$$\frac{,\ \vdash A :: K}{,\ \vdash \texttt{<=}A\texttt{::}K\texttt{>} :: \Omega} \qquad \text{(C-TRANS)}$$

$$\frac{,\ \vdash x\rho \Rightarrow \texttt{<}K\texttt{>}}{,\ \vdash x\rho! :: K} \qquad \text{(C-EXT-O)}$$

$$\frac{,\ \vdash x\rho \Rightarrow \texttt{<=}A\texttt{::}K\texttt{>}}{,\ \vdash x\rho! :: K} \qquad \text{(C-EXT-T)}$$

$$\frac{\vdash\ ,\ \ \mathit{valid}}{,\ \vdash \mathbf{rec} :: (\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega} \qquad \text{(C-REC)}$$

$$\frac{\vdash\ ,\ \ \mathit{valid}}{,\ \vdash \mathbf{ref} :: \Omega{\Rightarrow}\Omega} \qquad \text{(C-REF)}$$

Definition 6.3.3 (Constructor Equality Rules)

$$\frac{,\ \vdash A :: K}{,\ \vdash A = A :: K} \qquad \text{(E-REFL)}$$

$$\frac{,\ \vdash A' = A :: K}{,\ \vdash A = A' :: K} \qquad \text{(E-SYM)}$$

$$\frac{,\ \vdash A = A' :: K \qquad ,\ \vdash A' = A'' :: K}{,\ \vdash A = A'' :: K} \qquad \text{(E-TRAN)}$$

$$\frac{\begin{array}{c},\ \vdash A_1 = A_1' :: \Omega \\ ,\ , x{:}A_1 \vdash A_2 = A_2' :: \Omega\end{array}}{,\ \vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega} \qquad \text{(E-DFUN)}$$

$$\frac{\begin{array}{c},\ \vdash A_1 = A_1' :: \Omega \\ ,\ , x{:}A_1 \vdash A_2 = A_2' :: \Omega\end{array}}{,\ \vdash \Sigma x{:}A_1.\, A_2 = \Sigma x{:}A_1'.\, A_2' :: \Omega} \qquad \text{(E-DSUM)}$$

$$\frac{,\ , \alpha{::}K \vdash A = A' :: K'}{,\ \vdash \lambda\alpha{::}K.\, A = \lambda\alpha{::}K.\, A' :: K{\Rightarrow}K'} \qquad \text{(E-LAM)}$$

$$\frac{\begin{array}{c},\ \vdash A_2 = A_2' :: K \\ ,\ \vdash A_1 = A_1' :: K{\Rightarrow}K'\end{array}}{,\ \vdash A_1\, A_2 = A_1'\, A_2' :: K'} \qquad \text{(E-APP)}$$

$$\frac{,\ \vdash A = A' :: K}{,\ \vdash \texttt{<=}A\texttt{::}K\texttt{>} = \texttt{<=}A'\texttt{::}K\texttt{>} :: \Omega} \qquad \text{(E-TRANS)}$$

$$\frac{,\ , \alpha\texttt{::}K \vdash A :: K' \qquad ,\ \vdash A' :: K}{,\ \vdash (\lambda\alpha\texttt{::}K.\, A)\, A' = [A'/\alpha]A :: K'} \qquad \text{(E-BETA)}$$

$$\frac{,\ \vdash A :: K \Rightarrow K' \qquad \alpha \notin FCV(A)}{,\ \vdash \lambda\alpha\texttt{::}K.\, A\, \alpha = A :: K \Rightarrow K'} \qquad \text{(E-ETA)}$$

$$\frac{,\ \vdash x\rho! :: K \qquad ,\ \vdash x\rho \Rightarrow A \qquad ,\ \vdash A = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega}{,\ \vdash x\rho! = A' :: K} \qquad \text{(E-ABBREV)}$$

Definition 6.3.4 (Place Lookup Rules)

$$\frac{\vdash ,\ \mathit{valid} \qquad x{:}A \in ,}{,\ \vdash x \Rightarrow A} \qquad \text{(P-INIT)}$$

$$\frac{,\ \vdash x\rho \Rightarrow A \qquad ,\ \vdash A = A' :: \Omega \qquad A'' = \mathcal{S}(A', x\rho, \rho')}{,\ \vdash x\rho\rho' \Rightarrow A''} \qquad \text{(P-MOVE)}$$

I have designed these rules and those of all the other judgments I shall discuss later so that the following *structural property* holds: all the sub-components of a judgment are either valid or well-typed as appropriate. For example, if $,\ \vdash A_1 = A_2 :: K$ then it will be true that $\vdash ,$ valid, $,\ \vdash A_1 :: K$, and $,\ \vdash A_2 :: K$. This result is immediate for assignments:

Lemma 6.3.5 (Structural property)

1. *A derivation of* $\vdash ,\ , D$ *valid contains* $\vdash ,$ *valid as a sub-derivation.*

2. *A derivation of* $,\ \vdash A :: K$ *contains* $\vdash ,$ *valid as a sub-derivation.*

3. *A derivation of* $,\ \vdash A_1 = A_2 :: K$ *contains* $\vdash ,$ *valid as a sub-derivation.*

4. *A derivation of* $,\ \vdash x\rho \Rightarrow A$ *contains* $\vdash ,$ *valid as a sub-derivation.*

Proof: By simultaneous structural induction on the typing derivations. □

The fact that $\vdash ,$ valid is actually a sub-derivation is useful when using proof by induction on typing derivations. Note that because valid assignments are required not to redeclare variables, this result means that in rules such as C-DFUN we must have that $x \notin \mathrm{dom}(,)$.

The structural property is much harder to establish for constructors and terms and is proved much later. This fact results from a design choice: I could have added extra conditions to the rules that would have made those results immediate as well. However, this choice would have just pushed the work involved elsewhere; it is not possible, for example, to avoid proving that β-reduction on constructors preserves validity. I did not have time to explore the full consequences of adding such conditions; it is possible that a careful choice could lead to a somewhat simpler proof.

The C-OPAQ and C-TRANS rules check the validity of reified constructor types in a straightforward manner. The validity of occurrences of constructor extractions are checked by the C-EXT-O and C-EXT-T rules. A constructor extraction $x\rho!$ has kind K only if the place $x\rho$ points to a reified constructor of kind K. There are two separate rules because the place lookup judgment does not incorporate a notion of subsumption; this fact means that the cases where $x\rho$ has types $<K>$ and $<=A::K>$ must be handled separately. (If place lookup handled subsumption, only C-EXT-O would be needed because $<=A::K>$ is a subtype of $<K>$.)

The E-ABBREV rule handles equality for $x\rho!$. I chose a version of this rule that does not require that the kind of the constructor extracted from $x\rho$ (K') matches the kind assigned originally to $x\rho!$ (K). Later on in the proof, I establish that these two kinds must be the same because equality preserves kind information for constructor components. (E.g., if $\Gamma \vdash <=A::K> = <=A'::K'> :: \Omega$ then $K = K'$.) The extra equality step on the type of $x\rho$ before it is used is there to mirror the relevant rewriting rule (R-ABBREV, defined in Section 7.3).

Alternatively, I could have chosen a version of the rule that requires the two kinds to be the same; for example, I could have removed the $\Gamma \vdash x\rho! :: K$ precondition and replaced K' by K. It would still be necessary to prove that the actual extracted constructor has the right kind; the point where it would have to be proved would change however. In hindsight, this change seems unlikely to have much effect on the complexity of the proofs. Because of this fact and the fact that the alternative version is simpler and more elegant, I think it would have been a better choice.

6.4 Place Lookup

As I discussed earlier, the place lookup judgment is used to determine the type of terms of the form $x\rho$ (i.e., places) that appear in constructors; the place lookup judgment is specialized for this purpose and does not handle subsumption. Because subtyping in my system only forgets information about constructor components, it cannot change what kind $x\rho!$ is or what it is equal to; this fact means that place lookup can safely disregard subsumption, replacing it with the more limited ability to allow a term's type to be replaced by an equal type.

If we consider just the typing rules that apply to terms of the form $x\rho$, replacing subsumption as discussed above and using old-style extended-value rules, we might have something like the following:

$$\frac{\vdash ,\ \text{valid} \qquad x{:}A \in ,}{,\ \vdash x : A} \tag{VAR}$$

$$\frac{,\ \vdash x\rho : A \qquad ,\ \vdash A = A' :: \Omega}{,\ \vdash x\rho : A'} \tag{EQ}$$

$$\frac{,\ \vdash x\rho : \Sigma x'{:}A_1.\, A_2}{,\ \vdash x\rho.1 : A_1} \tag{FST}$$

$$\frac{,\ \vdash x\rho : \Sigma x'{:}A_1.\, A_2 \qquad x' \notin \mathrm{FTV}(A_2)}{,\ \vdash x\rho.2 : A_2} \tag{SND}$$

$$\frac{,\ \vdash x\rho : {<}K{>}}{,\ \vdash x\rho : {<}{=}x\rho{::}K{>}} \tag{RC-VAL}$$

$$\frac{,\ \vdash x\rho : \Sigma x'{:}A_1.\, A_2 \qquad ,\ \vdash x\rho.1 : A_1'}{,\ \vdash x\rho : \Sigma x'{:}A_1'.\, A_2} \tag{FST-VAL}$$

$$\frac{,\ \vdash x\rho : \Sigma x'{:}A_1.\, A_2 \qquad ,\ \vdash x\rho.2 : A_2'}{,\ \vdash x\rho : \Sigma x'{:}A_1.\, A_2'} \tag{SND-VAL}$$

(The FST-VAL and SND-VAL rules are used to recursively apply the RC-VAL rule to the sub-components of dependent sums.)

By taking advantage of the fact that places may appear in constructors, we can improve on these rules by replacing SND with a simpler rule:

$$\frac{,\ \vdash x\rho : \Sigma x'{:}A_1.\, A_2}{,\ \vdash x\rho.2 : [x\rho.1/x']A_2} \tag{SND2}$$

Here $[x\rho.1/x']$ denotes the *place substitution* where $x\rho.1$ is substituted for x' (defined below on places, constructors, declarations, and assignments). In essence, SND2 is just the standard elimination rule for dependent sums limited to $x\rho$'s. It can be derived from the SND, EQ, RC-VAL, FST-VAL, and SND-VAL rules (see Section 4.7).

Definition 6.4.1 (Place substitution)

$$\begin{array}{rcll}
[x\rho/x']x'\rho' & = & x\rho\rho' & \\
[x\rho/x']x''\rho'' & = & x''\rho'' & (x' \neq x'') \\
\\
[x\rho/x'](x''\rho''!) & = & ([x\rho/x']x''\rho'')! & \\
[x\rho/x']\Pi x''{:}A_1.\, A_2 & = & \Pi x''{:}[x\rho/x']A_1.\, [x\rho/x']A_2 & (x'' \neq x,\ x'' \neq x') \\
[x\rho/x']\Sigma x''{:}A_1.\, A_2 & = & \Sigma x''{:}[x\rho/x']A_1.\, [x\rho/x']A_2 & (x'' \neq x,\ x'' \neq x') \\
[x\rho/x']\lambda\alpha{::}K.\, A & = & \lambda\alpha{::}K.\, [x\rho/x']A & \\
[x\rho/x'](A_1\, A_2) & = & [x\rho/x']A_1\, [x\rho/x']A_2 & \\
[x\rho/x']{<}{=}A'{::}K{>} & = & {<}{=}[x\rho/x']A'{::}K{>} & \\
[x\rho/x']\alpha & = & \alpha & \\
[x\rho/x']{<}K{>} & = & {<}K{>} & \\
[x\rho/x']\mathbf{rec} & = & \mathbf{rec} & \\
[x\rho/x']\mathbf{ref} & = & \mathbf{ref} & \\
\\
[x\rho/x'](\alpha{::}K) & = & \alpha{::}K & \\
[x\rho/x'](x''{:}A) & = & x''{:}[x\rho/x']A & \\
\\
[x\rho/x']\bullet & = & \bullet & \\
[x\rho/x'](,\ , D) & = & ([x\rho/x'],\ ), [x\rho/x']D &
\end{array}$$

After we replace SND with SND2, the VAL rules are no longer needed and can be discarded: While the VAL rules do allow more types to be derived for $x\rho$, they do not allow more kinds or equations for $x\rho!$ to be derived. (For example, if $x{:}{<}\Omega{>}$, we can derive $x{:}{<}{=}x!{::}\Omega{>}$ using RC-VAL resulting in $x! = x!$, a redundant equation given E-REFL.)

By introducing a *selection function* ($\mathcal{S}(A, x\rho, \rho')$), we can combine the FST and SND2 rules into a more general rule:

$$\frac{,\ \vdash x\rho : A \qquad A' = \mathcal{S}(A, x\rho, \rho')}{,\ \vdash x\rho\rho' : A'} \tag{MOVE}$$

Definition 6.4.2 (Selection)

$$\begin{array}{rcl}
\mathcal{S}(A, x\rho, \epsilon) & = & A \\
\\
\mathcal{S}(\Sigma x'{:}A_1.\, A_2, x\rho, .1\rho') & = & \mathcal{S}(A_1, x\rho.1, \rho') \\
\mathcal{S}(\Sigma x'{:}A_1.\, A_2, x\rho, .2\rho') & = & \mathcal{S}([x\rho.1/x']A_2, x\rho.2, \rho')
\end{array}$$

The selection function $\mathcal{S}(A, x\rho, \rho')$ computes the type (if any) for $x\rho\rho'$ that can be inferred from $\Gamma \vdash x\rho : A$ using just the FST and SND2 rules. It does this computation by descending into the type A according to the path ρ': at each step, if the next component of ρ' is .1, it selects the first component (if any) of A, and if the next component is .2, it selects the second component (if any) of A. Because the second component can depend on the first component, when selecting the second component the selection function must replace the internal name for the first component (e.g., x' in $\Sigma x'{:}A_1.\, A_2$) with its external name $x\rho.1$, where $x\rho$ is the external name for A. In order that the selection function can do this replacement, it is passed A's external name as its second argument.

Thus, if ρ' is set to .1 or to .2, then MOVE is the same as FST or SND2 respectively. In the general case, one application of MOVE is equivalent to a series of FST and SND2 applications.

The place lookup judgment implements this improved version of the rules: the P-INIT rule implements VAR and the P-MOVE rule implements EQ followed by MOVE. I designed the place lookup judgment to closely mirror the relevant rewriting rules (R-EXT and R-ABBREV, Section 7.3); in particular, $x\rho$'s type is obtained by alternating the use of equality (to get the type into the right form) and the selection function (to get the type of a subcomponent).

An alternative design I considered was to use a more restrictive rule that only permitted the use of equality once followed by the use of selection once. Given confluence and its associated results, it can be shown that this design is functionally equivalent to the one I used: both lead to the same types for $x\rho$'s. Unfortunately, under the alternate design this equivalence is required to prove confluence; a mutual dependency thus results in the proofs which prevents this design from working.

6.5 Basic Propositions

In this section, I prove a number of miscellaneous basic propositions for use later on. Included are some propositions on the properties of the basic operators (substitution, selection, and the domain function), as well as a series of propositions showing that all term and constructor variables in judgments are bound.

Lemma 6.5.1 $[x\rho/x']x''\rho'\rho'' = ([x\rho/x']x''\rho')\rho''$.

Lemma 6.5.2 *if* $\mathcal{S}(A, x\rho, .i\rho')$ *or* $\mathcal{S}(\mathcal{S}(A, x\rho, .i), x\rho.i, \rho')$ *exists then*

$$\mathcal{S}(A, x\rho, .i\rho') = \mathcal{S}(\mathcal{S}(A, x\rho, .i), x\rho.i, \rho')$$

Proof: By inspection of the definition of selection. □

Lemma 6.5.3 *If $\mathcal{S}(A, x\rho_1, \rho_2\rho_3)$ exists then*

$$\mathcal{S}(A, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(A, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3)$$

Proof: By structural induction on ρ_2. The non-basis case is as follows:

$$\begin{array}{lll} & \mathcal{S}(A, x\rho_1, .i\rho\rho_3) & \\ = & \mathcal{S}(\mathcal{S}(A, x\rho_1, .i), x\rho_1.i, \rho\rho_3) & \text{(Lemma 6.5.2)} \\ = & \mathcal{S}(\mathcal{S}(\mathcal{S}(A, x\rho_1, .i), x\rho_1.i, \rho), x\rho_1.i\rho, \rho_3) & \text{(induction hypothesis)} \\ = & \mathcal{S}(\mathcal{S}(A, x\rho_1, .i\rho), x\rho_1.i\rho, \rho_3) & \text{(Lemma 6.5.2)} \end{array}$$

□

Lemma 6.5.4 (Assignment properties)

1. *$dom(, ; , ') = dom(,) \cup dom(, ')$*
2. *If $x{:}A \in ,$ then $x \in dom(,)$, $FCV(A) \subseteq FCV(,)$, and $FTV(A) \subseteq FTV(,)$.*

Lemma 6.5.5 (Free term variables)

1. *$FTV([A'/\alpha]A) \subseteq FTV(A) \cup FTV(A')$*
2. *$FTV([x\rho/x']A) \subseteq \{x\} \cup (FTV(A) \Leftrightarrow \{x'\})$*
3. *If $\mathcal{S}(A, x\rho, \rho')$ exists then $FTV(\mathcal{S}(A, x\rho, \rho')) \subseteq \{x\} \cup FTV(A)$.*

Theorem 6.5.6 (Scoping of term variables)

1. *If $, \vdash x\rho \Rightarrow A$ then $x \in dom(,)$.*
2. *If $, \vdash A :: K$ then $FTV(A) \subseteq dom(,)$.*
3. *If $, \vdash A_1 = A_2 :: K$ then $\forall i.\ FTV(A_i) \subseteq dom(,)$.*
4. *If $, \vdash x\rho \Rightarrow A$ then $FTV(A) \subseteq dom(,)$.*

Proof: Proved sequentially by structural induction on the typing derivations, using Lemmas 6.5.5 and 6.3.5 as needed. □

Corollary 6.5.7

1. *If* $\vdash$, *valid then* $FTV(,\ ) \subseteq dom(,\ )$.
2. *If* $\vdash$, , $x{:}A$ *valid then* $x \notin FTV(,\ )$.

Proof: The first part is proved using structural induction on the typing derivations, using Theorem 6.5.6 as needed. The second part follows almost immediately from the first part and the rule DECL-T. □

Theorem 6.5.8 (Scoping of constructor variables)

1. *If* , $\vdash A :: K$ *then* $FCV(A) \subseteq dom(,\ )$.
2. *If* $\vdash$, *valid then* $FCV(,\ ) \subseteq dom(,\ )$.

Proof: Proved sequentially using structural induction on the derivations applying Lemma 6.3.5 as needed. □

Corollary 6.5.9 *If* $\vdash$, , $\alpha{::}K$ *valid then* $\alpha \notin FCV(,\ )$.

Lemma 6.5.10 *If* $\vdash\ ,_1;\ ,_2$ *valid then* $dom(,_1) \cap dom(,_2) = \emptyset$.

Proof: Inspection of the typing rules for assignments combined with the results from Lemma 6.3.5 reveal that valid assignments never redeclare variables. □

6.6 Tagging

In the next section I introduce the tagged system which differs from the (untagged) kernel system by having tagged places in constructors and by having slightly different operators and type rules. Throughout these proofs, I shall be dealing with a number of different systems. I shall use the same notation for each system whenever possible. Thus, A shall denote a constructor in each system rather than A denoting an untagged constructor, B a tagged one, C an erased tagged one, and so on. This convention makes it easy to deal with systems that differ only slightly from each other; it also means that there are fewer syntactic categories that need to be remembered.

Which system is meant shall either be clear from context or be explicitly stated (e.g., "... a tagged constructor A ..."). Because notation occurs most frequently in propositions and definitions, I shall use a special notation to specify what system the variables in a proposition or definition are from. The variables' system shall be indicated by placing its name in square brackets at the beginning of the proposition or definition; if no system is indicated, then the variables are from the original kernel system. In the rare cases where variables from multiple systems are needed, the system name shall be replaced with the word "mixed" and the system of each variable shall be explicitly stated. The system of the operators and judgments shall be clear from context.

Lemma [Tagged] 6.6.1 (Notation Example)
If $\vdash$, *valid then* $\vdash$, $^{\ominus}$ *valid.*

Here, the variables' system is the tagged system so , is an assignment from the tagged system. Because tag removal ($\Leftrightarrow^{\ominus}$, defined later in Section 9.1) is an operator that converts objects to the untagged system, , $^{\ominus}$ must be an assignment from the untagged system, the first judgment must be a tagged one, and the second one must be an untagged one. The lemma thus states that a valid tagged assignment may be converted to a valid untagged assignment by tag removal. I shall prove this result later as part of Theorem 9.3.1.

I originally added tags by replacing $x\rho!$ in the definition of constructor syntax with $x_A\rho!$. I also had to alter place substitution so that it replaces a term variable with a tagged place:

$$[x_A\rho/x']x'_{A'}\rho' = x_A\rho\rho'$$

This change was necessary so that the place substitution function would have a type that it could use as a tag for x. (The type of x is not computable from the type of x' or $x\rho$ due to missing information.) I made a similar change to the selection function.

Although I was able to work out much of the proof using this definition, I eventually ran into a fatal problem with proving confluence. The problem arises from the fact that under this definition the place substitution operator "replaces" one tag with another one (e.g., A for A' in the previous equation). If the constructor being rewritten is valid, this replacement will result in a constructor that has a common reduct with the original one. However, it may take many steps to reach the common reduct. This fact allows a counterexample to the diamond property to be constructed.

Consider the constructor $x_A.2!$ where $A = \Sigma x'{:}A_9.{<}{=}x'_{A_0}!{::}\Omega{>}$, $A_0 = {<}{=}\alpha{::}\Omega{>}$, and A_9 is such that it rewrites to A_0, but does not rewrite to ${<}{=}A'{::}K{>}$ for any K or A' in less than nine steps. Starting from this constructor, in a single step we could extract the constructor α from the sub-constructor $x'_{A_0}!$ or we could extract the constructor from $x_A.2!$ yielding $[x_A.1/x'](x'_{A_0}!) = x_A.1!$. Thus, in one step we can reach both of the following:

$$x_{\Sigma x':A_9.<=\alpha::\Omega>}.2! \qquad\qquad x_{\Sigma x':A_9.<=x'_{A_0}!::\Omega>}.1!$$

The only way to make these two constructors converge again by rewriting is by reducing both of them to α. Unfortunately, because of the way A_9 was constructed, this reduction will take at least nine steps because A_9 needs to be reduced to $<=A'::K>$ for some K and A' before the second constructor as a whole can be reduced. Thus, the original constructor is a counterexample to the diamond property because we can take one step away from it in two different ways yet be unable to rejoin within one step.

I solved this problem by switching to *floating tags*. Whereas, before, tags had to be placed immediately after the term variable (i.e., $x_A\rho!$), with floating tags the tag may be placed anywhere in the path following the term variable (i.e., $x\rho_A\rho'!$). The type indicated by a floating tag is that of the preceding place. Thus, if , $\vdash x\rho_A\rho'! :: K$ then , $\vdash x\rho \Rightarrow A$. This change means that the place substitution operator no longer needs to do tag "replacement" or substitute tagged places because it can continue to use the same tag as before, just in a possibly new location:

$$[x\rho/x'](x''\rho'_A\rho''!) = ([x\rho/x']x''\rho')_{[x\rho/x']A}\rho''!$$

6.7 The Tagged System

Except as noted in this section, the tagged system is identical to the untagged one. The only change in syntax is to replace $x\rho!$ in the definition of constructors with $x\rho_A\rho'!$. Constructor extractions in the tagged system are accordingly $x\rho_A\rho'!$'s. I shall call $x\rho_A\rho'$'s *tagged places*, reserving the term place to always refer to $x\rho$'s.

The operator definitions have to be changed slightly to deal with the changed syntax:

Definition [Tagged] 6.7.1 (Operator changes)

$$\begin{aligned}
FCV(x\rho_A\rho'!) &= FCV(A) \\
FTV(x\rho_A\rho'!) &= FTV(A) \cup \{x\} \\
[A/\alpha]x\rho_{A'}\rho'! &= x\rho_{[A/\alpha]A'}\rho'! \\
[x\rho/x'](x''\rho'_A\rho''!) &= ([x\rho/x']x''\rho')_{[x\rho/x']A}\rho''!
\end{aligned}$$

The operators remain defined as before for all other inputs.

The C-EXT-O, C-EXT-T, and E-ABBREV rules are replaced by the following new rules:

Definition [Tagged] 6.7.2 (Rule changes)

$$\frac{, \vdash x\rho\rho' \Rightarrow \texttt{<}K\texttt{>} \qquad , \vdash x\rho \Rightarrow A}{, \vdash x\rho_{A}\rho'! :: K} \qquad \text{(C-EXT-O2)}$$

$$\frac{, \vdash x\rho\rho' \Rightarrow \texttt{<=}A'\texttt{::}K\texttt{>} \qquad , \vdash x\rho \Rightarrow A}{, \vdash x\rho_{A}\rho'! :: K} \qquad \text{(C-EXT-T2)}$$

$$\frac{\begin{array}{c}, \vdash x\rho_{A_1}\rho'\rho''! :: K \\ , \vdash A_1 = A :: \Omega \qquad A_2 = \mathcal{S}(A, x\rho, \rho')\end{array}}{, \vdash x\rho_{A_1}\rho'\rho''! = x\rho\rho'_{A_2}\rho''! :: K} \qquad \text{(E-EXT2)}$$

$$\frac{, \vdash x\rho_{A}! :: K \qquad , \vdash A = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega}{, \vdash x\rho_{A}! = A' :: K} \qquad \text{(E-ABBREV2)}$$

These tagged rules are designed similarly to the untagged rules so that they possess the same structural property. C-EXT-O2 and C-EXT-T2 are essentially the same as C-EXT-O and C-EXT-T respectively except that I added an extra precondition ($, \vdash x\rho \Rightarrow A$) to ensure that the tag (A) assigned to $x\rho$ is consistent with , .

E-ABBREV2 and the new E-EXT-2 rule are designed to mirror closely the relevant rewriting rules:

$$\frac{A_1 \rightarrow A \qquad A_2 = \mathcal{S}(A, x\rho, \rho')}{x\rho_{A_1}\rho'\rho''! \rightarrow x\rho\rho'_{A_2}\rho''!} \qquad \text{(R-EXT)}$$

$$\frac{A \rightarrow \texttt{<=}A'\texttt{::}K'\texttt{>}}{x\rho_{A}! \rightarrow A'} \qquad \text{(R-ABBREV)}$$

(R-EXT is used both to rewrite the tag in place and to move it to the right while R-ABBREV is used to perform the extraction once the tag has been reduced to a <=A'::K> type; see Section 7.3 for more.)

The effect of the old E-ABBREV rule must be gotten by using repeated applications of E-EXT-2 followed by an application of E-ABBREV2 (connected using E-TRAN); more applications are needed because the tagged system, aside from E-TRAN, has only simple equality rules, which can only do a small amount of work each time they are applied. Like E-ABBREV, E-ABBREV2 also does not require that the kind of the constructor extracted from a constructor extraction match the kind assigned originally to that constructor extraction.

6.8 Basic Tagged Propositions

In this section, I prove a number of miscellaneous basic propositions about the tagged system for use later on. Most of these propositions are tagged versions of those I proved previously for the untagged system in Section 6.5. This duplication is necessary because these propositions are needed in order to prove that results can be transfered back and forth between the two systems (see Sections 9.1–9.3). I shall point out the new propositions as I come to them.

Lemma [Tagged] 6.8.1 *If* $\mathcal{S}(A, x\rho_1, \rho_2\rho_3)$ *exists then*

$$\mathcal{S}(A, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(A, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3)$$

Proof: Exactly the same as the untagged version (Lemma 6.5.3). □

Lemma [Tagged] 6.8.2 (Assignment properties)

1. $dom(\, ;\, ') = dom(\,) \cup dom(\, ')$
2. *If* $x{:}A \in\ $, *then* $x \in dom(\,)$, $FCV(A) \subseteq FCV(\,)$, *and* $FTV(A) \subseteq FTV(\,)$.

Lemma [Tagged] 6.8.3 (Structural property)

1. *A derivation of* $\vdash\ ,\ , D$ *valid contains* $\vdash\ ,$ *valid as a sub-derivation.*
2. *A derivation of* $, \vdash A :: K$ *contains* $\vdash\ ,$ *valid as a sub-derivation.*
3. *A derivation of* $, \vdash x\rho \Rightarrow A$ *contains* $\vdash\ ,$ *valid as a sub-derivation.*
4. *A derivation of* $, \vdash A_1 = A_2 :: K$ *contains* $\vdash\ ,$ *valid as a sub-derivation.*

Proof: By simultaneous structural induction on the typing derivations. □

Lemma [Tagged] 6.8.4 (Free term variables)

1. $FTV([A'/\alpha]A) \subseteq FTV(A) \cup FTV(A')$
2. $FTV([x\rho/x']A) \subseteq \{x\} \cup (FTV(A) \Leftrightarrow \{x'\})$

3. *If $\mathcal{S}(A, x\rho, \rho')$ exists then $FTV(\mathcal{S}(A, x\rho, \rho')) \subseteq \{x\} \cup FTV(A)$.*

Theorem [Tagged] 6.8.5 (Scoping of term variables)

1. *If $\vdash$, valid then $FTV(\, ,\,) \subseteq dom(\, ,\,)$.*
2. *If , $\vdash A :: K$ then $FTV(A) \subseteq dom(\, ,\,)$.*
3. *If , $\vdash x\rho \Rightarrow A$ then $x \in dom(\, ,\,)$ and $FTV(A) \subseteq dom(\, ,\,)$.*
4. *If , $\vdash A_1 = A_2 :: K$ then $\forall i.\ FTV(A_i) \subseteq dom(\, ,\,)$.*

Proof: Proved by simultaneous structural induction on the derivations applying Lemmas 6.8.3 and 6.5.5 as needed. Sample cases:

DECL-T: Given $\vdash$, , $x{:}A$ valid derived via rule DECL-T from , $\vdash A :: \Omega$ and $x \notin$ dom(,). By Lemma 6.8.3, $\vdash$, valid. Applying the induction hypothesis then gives us that FTV(A) $\subseteq$ dom(,) and FTV(,) $\subseteq$ dom(,). Since FTV(, , $x{:}A$) = FTV(,) $\cup$ FTV(A) and dom(, , $x{:}A$) = dom(,) $\cup \{x\}$, this means that FTV(, , $x{:}A$) $\subseteq$ dom(,) $\subseteq$ dom(, , $x{:}A$).

C-DFUN: Given , $\vdash \Pi x{:}A.\, A' :: \Omega$ derived via rule C-DFUN from , , $x{:}A \vdash A' :: \Omega$. By Lemma 6.8.3, $\vdash$, , $x{:}A$ valid $\Rightarrow$, $\vdash A :: \Omega$ (DECL-T). Applying the induction hypothesis then gives us that FTV(A) $\subseteq$ dom(,) and FTV(A') $\subseteq$ dom(, , $x{:}A$) = dom(,) $\cup \{x\} \Rightarrow$ (FTV(A') $\Leftrightarrow \{x\}$) $\subseteq$ dom(,). Hence, FTV($\Pi x{:}A.\, A'$) = FTV(A) $\cup$ (FTV(A') $\Leftrightarrow \{x\}$) $\subseteq$ dom(,).

P-INIT: Given , $\vdash x \Rightarrow A$ derived via rule P-INIT from $\vdash$, valid and $x{:}A \in$, . Applying the induction hypothesis gives us that FTV(,) $\subseteq$ dom(,). By Lemma 6.8.2, $x \in$ dom(,) and FTV(A) $\subseteq$ FTV(,) $\subseteq$ dom(,).

E-BETA: Given , $\vdash (\lambda\alpha{::}K.\, A)\, A' = [A'/\alpha]A :: K'$ derived via rule E-BETA from , , $\alpha{::}K \vdash A :: K'$ and , $\vdash A' :: K$. Applying the induction hypothesis gives us that FTV(A') $\subseteq$ dom(,) and FTV(A) $\subseteq$ dom(, , $\alpha{::}K$) = dom(,) $\cup \{\alpha\} \Rightarrow$ FTV(A) $\subseteq$ dom(,) (α is not a term variable). By Lemma 6.8.4, FTV($[A'/\alpha]A$) $\subseteq$ FTV(A) $\cup$ FTV(A') $\subseteq$ dom(,). Finally, FTV($(\lambda\alpha{::}K.\, A)\, A'$) = FTV($A$)$\cup$FTV($A'$) $\subseteq$ dom(,).

E-EXT2: Given , $\vdash x\rho_{A_1}\rho'\rho''! = x\rho\rho'_{A_2}\rho''! :: K$ derived via rule E-EXT2 from , $\vdash x\rho_{A_1}\rho'\rho''! :: K$, , $\vdash A_1 = A :: \Omega$, and $\mathcal{S}(A, x\rho, \rho') = A_2$. Applying the induction hypothesis gives us that FTV($x\rho_{A_1}\rho'\rho''!$) = $\{x\} \cup$ FTV(A_1) $\subseteq$ dom(,) and FTV(A) $\subseteq$ dom(,). By Lemma 6.8.4, FTV(A_2) = FTV($\mathcal{S}(A, x\rho, \rho')$) $\subseteq$ $\{x\} \cup$ FTV(A) $\subseteq$ dom(,). Finally, FTV($x\rho\rho'_{A_2}\rho''!$) = $\{x\} \cup$ FTV(A_2) $\subseteq$ dom(,).

□

For the untagged system, I needed to show only that all constructor variables in valid assignment and constructor judgments are bound. I shall need this result for all four judgments in the tagged case. Accordingly, I have extended the relevant theorem (Theorem 6.8.7 here) and added the following new lemma to help handle the equality judgment case:

Lemma [Tagged] 6.8.6 (Free constructor variables)

1. *FCV*$([A'/\alpha]A) \subseteq$ *FCV*$(A') \cup ($*FCV*$(A) \Leftrightarrow \{\alpha\})$
2. *FCV*$([x\rho/x']A) =$ *FCV*(A)
3. *If* $\mathcal{S}(A, x\rho, \rho')$ *exists then FCV*$(\mathcal{S}(A, x\rho, \rho')) \subseteq$ *FCV*(A).

Theorem [Tagged] 6.8.7 (Scoping of constructor variables)

1. *If* $,\ \vdash A :: K$ *then FCV*$(A) \subseteq$ *dom*$(,\ )$.
2. *If* $,\ \vdash x\rho \Rightarrow A$ *then FCV*$(A) \subseteq$ *dom*$(,\ )$.
3. *If* $,\ \vdash A_1 = A_2 :: K$ *then FCV*$(A_1) \subseteq$ *dom*$(,\ )$ *and FCV*$(A_2) \subseteq$ *dom*$(,\ )$.

Proof: Proved by simultaneous structural induction on the derivations applying Lemmas 6.8.3 and 6.8.6 as needed. Sample cases:

C-DSUM: Applying Lemma 6.8.3 to the given $,\ , x{:}A \vdash A' :: \Omega$, gives us that $,\ \vdash A :: \Omega$. Applying the induction hypothesis twice, we have that FCV$(A) \subseteq$ dom$(,\ )$ and that FCV$(A') \subseteq$ dom$(,\ , x{:}A) =$ dom$(,\ ) \cup \{x\}$. Since x is not a constructor variable, we more precisely have FCV$(A') \subseteq$ dom$(,\ )$. Thus, FCV$(\Sigma x{:}A.\, A') =$ FCV$(A) \cup$ FCV$(A') \subseteq$ dom$(,\ )$.

P-INIT: Repeated use of Lemma 6.3.5 shows that $\vdash\ ,\ ', x{:}A$ valid for some prefix $,\ '$ of $,\ $. Hence, by DECL-T, $,\ ' \vdash A :: \Omega$. Applying the induction hypothesis, then gives us that FCV$(A) \subseteq$ dom$(,\ ')$. By Lemma 6.8.2, we have that dom$(,\ ') \subseteq$ dom$(,\ )$ so that FCV$(A) \subseteq$ dom$(,\ )$.

P-MOVE: By applying the induction hypothesis, we have that FCV$(A') \subseteq$ dom$(,\ )$. By Lemma 6.8.6, FCV$(A'') =$ FCV$(\mathcal{S}(A', x\rho, \rho')) \supseteq$ FCV$(A') \supseteq$ dom$(,\ )$.

E-BETA: As in the C-DSUM case, applying Lemma 6.8.3 and the induction hypothesis, gives us that $\mathrm{FCV}(A') \subseteq \mathrm{dom}(,\ )$ and $\mathrm{FCV}(A) \subseteq \mathrm{dom}(,\ ) \cup \{\alpha\} \Rightarrow (\mathrm{FCV}(A) \Leftrightarrow \{\alpha\}) \subseteq \mathrm{dom}(,\ )$. By Lemma 6.8.6, $\mathrm{FCV}([A'/\alpha]A) \subseteq \mathrm{FCV}(A') \cup (\mathrm{FCV}(A) \Leftrightarrow \{\alpha\}) \subseteq \mathrm{dom}(,\ )$. By the definition of FCV(⇔), we also have that $\mathrm{FCV}((\lambda\alpha{::}K.\, A)\, A') = (\mathrm{FCV}(A) \Leftrightarrow \{\alpha\}) \cup \mathrm{FCV}(A') \subseteq \mathrm{dom}(,\ )$.

□

Theorem [Tagged] 6.8.8

1. *If* $\vdash ,\ $ *valid then* $FCV(,\ ) \subseteq dom(,\ )$.
2. *If* $\vdash ,\ , \alpha{::}K$ *valid then* $\alpha \notin FCV(,\ )$.

Proof: The first part is proved by structural induction on the derivation using Theorem 6.8.7 and Lemma 6.8.3 as needed. The second part follows almost immediately from the first part and rule DECL-C. □

Lemma [Tagged] 6.8.9 *If* $\vdash ,\ _1; ,\ _2$ *valid then* $dom(,\ _1) \cap dom(,\ _2) = \emptyset$.

Proof: Inspection of the typing rules for assignments combined with the results from Lemma 6.8.3 reveal that valid assignments never redeclare variables. □

The last two theorems are new. The first one shows that tagged judgments may be weakened by adding extra declarations to their assignment. The second one establishes some consequences of the fact that valid assignments never redeclare variables.

Theorem [Tagged] 6.8.10 (Weakening)
Suppose $\vdash ,\ ; ,\ '$ *valid and* $dom(,\ ') \cap dom(,\ '') = \emptyset$. *Then:*

1. *If* $\vdash ,\ ; ,\ ''$ *valid then* $\vdash ,\ ; ,\ '; ,\ ''$ *valid.*
2. *If* $,\ ; ,\ '' \vdash A :: K$ *then* $,\ ; ,\ '; ,\ '' \vdash A :: K$.
3. *If* $,\ ; ,\ '' \vdash x\rho \Rightarrow A$ *then* $,\ ; ,\ '; ,\ '' \vdash x\rho \Rightarrow A$.
4. *If* $,\ ; ,\ '' \vdash A_1 = A_2 :: K$ *then* $,\ ; ,\ '; ,\ '' \vdash A_1 = A_2 :: K$.

Proof: By simultaneous structural induction on the length of the derivations involving $,\ ; ,\ ''$. □

Lemma [Tagged] 6.8.11 (Valid assignment properties)
Suppose $\vdash ,$ *valid. Then:*

1. *If* $\alpha{::}K \in ,$ *and* $\alpha{::}K' \in ,$ *then* $K = K'$.

2. *If* $x{:}A \in ,$ *then* $, (x) = A$.

3. *If* $x \in$ *dom*$(,)$ *then* $, \vdash , (x) :: \Omega$.

Proof: Inspection of the typing rules for assignments combined with the results from Lemma 6.8.3 reveal that valid assignments never redeclare variables. The last part requires weakening plus repeated use of Lemma 6.8.3. □

Chapter 7

Types: Confluence

In this chapter, I introduce a rewriting relation for the tagged system and show that it is confluent and that it implements equality correctly. These results will be essential to producing a decision procedure for equality (needed for type checking) and proving soundness. I shall discuss the existence of normal forms under this relation and the decidability of the tagged judgments in the next chapter. All constructors, assignments, etc., in this chapter are tagged.

7.1 Rewriting

I shall write $A_1 \to_R A_2$ to denote that A_1 rewrites under relation R to A_2 in one step. If R is omitted, the standard rewriting relation of the system under consideration is meant. I shall use reduce as a synonym for rewrite, particularly when referring to rewriting steps (reductions) and to the results of rewriting steps (reducts). Some other useful definitions are as follows:

Definition [Tagged] 7.1.1 (Irreflexive rewriting) $A \to^1_R A'$ *iff* $A \to_R A'$ *and* $A \neq A'$.

Definition [Tagged] 7.1.2 (Multiple step rewriting) $A \to^*_R A'$ *iff* A *and* A' *are related by the reflexive and transitive closure of* $\to_R$.

Definition [Tagged] 7.1.3 (One or more step rewriting) $A \to^+_R A'$ *iff* A *and* A' *are related by the transitive closure of* $\to_R$.

Definition [Tagged] 7.1.4 (Conversion) $A \approx_R A'$ *iff* A *and* A' *are related by the reflexive, symmetric, and transitive closure of* $\to_R$.

Definition [Tagged] 7.1.5 (Bi-directional rewriting) *$A \rightleftharpoons_R A'$ iff $A \rightarrow_R A'$ or $A' \rightarrow_R A$.*

Definition [Tagged] 7.1.6 (Common reduct) *$A_1 \downarrow^*_R A_2$ iff $\exists A$ such that $A_1 \rightarrow^*_R A$ and $A_2 \rightarrow^*_R A$.*

7.2 The Proof Method

As I have discussed previously, equality in the tagged system does not depend on the assignment (modulo validity); this fact means that relatively standard methods can be used to prove confluence. I shall use the well known method of proving confluence by constructing a rewriting relation that possesses the *diamond property*:

Definition [Tagged] 7.2.1 (Diamond property)
$\rightarrow_R$ possesses the diamond property iff for all A, A_1, and A_2 such that $A \rightarrow_R A_1$ and $A \rightarrow_R A_2$ then $\exists A_3$ such that $A_1 \rightarrow_R A_3$ and $A_2 \rightarrow_R A_3$. This can be summarized by the following diagram, the form of which I shall use as a shorthand in proofs:

$$\begin{tikzcd}
& A \arrow[dl, "R"] \arrow[dr, "R"'] & \\
A_1 \arrow[dr, dotted, "R"'] & & A_2 \arrow[dl, dotted, "R"] \\
& A_3 &
\end{tikzcd}$$

Confluence of the rewriting relation follows directly from this property:

Theorem [Tagged] 7.2.2 (Confluence)
Suppose $\rightarrow$ possesses the diamond property. Then, if $A \rightarrow^* A_1$ and $A \rightarrow^* A_2$ then $\exists A'$ such that $A_1 \rightarrow^* A'$ and $A_2 \rightarrow^* A'$.

Proof: (Idea) Suppose A rewrites to A_1 in 2 steps and rewrites to A_2 in 3 steps. By using the diamond property repeatedly, we can "fill in the grid" to find an A' that both A_1 and A_2 rewrite to to as shown below:

$$\begin{tikzcd}
& & A \arrow[dl] \arrow[dr] & & & \\
& A_3 \arrow[dl] \arrow[dr, dotted] & & A_4 \arrow[dl, dotted] \arrow[dr] & & \\
A_1 \arrow[dr, dotted] & & A_6 \arrow[dl, dotted] \arrow[dr, dotted] & & A_5 \arrow[dl, dotted] \arrow[dr] & \\
& A_7 \arrow[dr, dotted] & & A_8 \arrow[dl, dotted] \arrow[dr, dotted] & & A_2 \arrow[dl, dotted] \\
& & A_9 \arrow[dr, dotted] & & A_{10} \arrow[dl, dotted] & \\
& & & A' & &
\end{tikzcd}$$

By using this method we can find an A' that both A_1 and A_2 rewrite to regardless of exactly how many steps it takes A to rewrite to A_1 or to A_2. ($A \rightarrow^* A'$ implies that A rewrites to A' in a finite number of steps by the definition of transitive closure.) $\square$

Because my system allows both β- and η-reductions, it only possesses confluence if attention is restricted to valid constructors. To see this fact, consider the following critical pair involving β and η:

$$\begin{tikzcd}
& \lambda\alpha{::}K.\,(\lambda\alpha'{::}K'.\,{<}\Omega{>})\,\alpha \arrow[dl, "\beta"'] \arrow[dr, "\eta"] & \\
\lambda\alpha{::}K.\,{<}\Omega{>} & & \lambda\alpha'{::}K'.\,{<}\Omega{>}
\end{tikzcd}$$

In order for us to have confluence, we must have that $K = K'$ (both reducts are in normal form). This equation will be true if $\lambda\alpha{::}K.\,(\lambda\alpha'{::}K'.\,{<}\Omega{>})\,\alpha$ is a valid constructor. If it is not a valid constructor, however, then K may not equal K' and confluence fails. This critical pair is the only point where we need to know we are dealing with valid constructors in order to show confluence; removing η-reduction from the system, for example, would allow proving confluence for arbitrary constructors.

This situation is normally handled by first proving that rewriting preserves the validity of constructors (i.e., subject reduction for constructors) and then proving a version of

Theorem 7.2.2 to which has been added the precondition that $\Gamma \vdash A :: K$. This method cannot be used here directly because in the presence of reified constructors proving subject reduction for constructors requires some form of confluence.

As an example, suppose we had $x{:}\texttt{<=}A'{::}K'\texttt{>} \vdash x_{\texttt{<=}A::K\texttt{>}}! :: K'$. Subject reduction then requires that $x_{\texttt{<=}A::K\texttt{>}}!$'s reduct ($A$ via extraction) have kind K'; since A has kind K, we must have that $K = K'$. Inspection of the relevant typing rules shows that we have $x{:}\texttt{<=}A'{::}K'\texttt{>} \vdash \texttt{<=}A{::}K\texttt{>} = \texttt{<=}A'{::}K'\texttt{>} :: \Omega$ which implies that $\texttt{<=}A{::}K\texttt{>} \approx \texttt{<=}A'{::}K'\texttt{>}$ because the rewriting relation implements equality. Showing that $K = K'$ from this fact requires some form of confluence in order to handle the possibility of β-expansions.

I shall use a technique due to Herman Geuvers [14] to handle this problem. The idea is to introduce a notion of *erasure* ($\Leftrightarrow^\circ$) which removes the kinds of constructor variables in constructor functions but not the kinds in reified constructor's types:

$$\begin{array}{rcl}(\lambda\alpha{::}K.\,A)^\circ & = & \lambda\alpha.\,A^\circ \\ \texttt{<=}A{::}K\texttt{>}^\circ & = & \texttt{<=}A^\circ{::}K\texttt{>}\end{array}$$

Because the resulting erased constructors do not have kinds attached to constructor-function arguments, the problematic $\beta\eta$-critical pair is not a problem for erased constructors. This fact means that confluence can be proved for arbitrary erased constructors; subject reduction is not required.

This confluence result can then be used to prove subject reduction. In the previous example we would first show that $\texttt{<=}A{::}K\texttt{>} \approx \texttt{<=}A'{::}K'\texttt{>}$ implies that $\texttt{<=}A{::}K\texttt{>}^\circ \approx \texttt{<=}A'{::}K'\texttt{>}^\circ \Rightarrow \texttt{<=}A^\circ{::}K\texttt{>} \approx \texttt{<=}A'^\circ{::}K'\texttt{>}$. By confluence on erased constructors, we then have that there exists an erased constructor B such that $\texttt{<=}A^\circ{::}K\texttt{>} \rightarrow^* B$ and $\texttt{<=}A'^\circ{::}K'\texttt{>} \rightarrow^* B$. Because $\texttt{<=}A_1{::}K_1\texttt{>}$ always rewrites to a constructor of form $\texttt{<=}A_2{::}K_1\texttt{>}$ for some A_2, we must have that $K = K'$ in the example as required.

Once subject reduction has been proved, confluence for the unerased constructors can be established using the normal method.

7.3 The Rewrite Relation

The rules for the rewriting relation are as follows:

Definition [Tagged] 7.3.1 (Rewrite relation)

$$A \rightarrow A \qquad \text{(R-REFL)}$$

$$\frac{A_1 \rightarrow A_1' \qquad A_2 \rightarrow A_2'}{\Pi x{:}A_1.\,A_2 \rightarrow \Pi x{:}A_1'.\,A_2'} \qquad \text{(R-DFUN)}$$

$$\frac{A_1 \rightarrow A_1' \qquad A_2 \rightarrow A_2'}{\Sigma x{:}A_1.\, A_2 \rightarrow \Sigma x{:}A_1'.\, A_2'} \qquad \text{(R-DSUM)}$$

$$\frac{A \rightarrow A'}{\lambda\alpha{::}K.\, A \rightarrow \lambda\alpha{::}K.\, A'} \qquad \text{(R-LAM)}$$

$$\frac{A_1 \rightarrow A_1' \qquad A_2 \rightarrow A_2'}{A_1\, A_2 \rightarrow A_1'\, A_2'} \qquad \text{(R-APP)}$$

$$\frac{A \rightarrow A'}{\texttt{<=}A{::}K\texttt{>} \rightarrow \texttt{<=}A'{::}K\texttt{>}} \qquad \text{(R-TRANS)}$$

$$\frac{A_1 \rightarrow A \qquad A_2 = \mathcal{S}(A, x\rho, \rho')}{x\rho_{A_1}\rho'\rho''! \rightarrow x\rho\rho'_{A_2}\rho''!} \qquad \text{(R-EXT)}$$

$$\frac{A_1 \rightarrow A_1' \qquad A_2 \rightarrow A_2'}{(\lambda\alpha{::}K.\, A_1)\, A_2 \rightarrow [A_2'/\alpha]A_1'} \qquad \text{(R-BETA)}$$

$$\frac{A \rightarrow A' \qquad \alpha \notin \mathit{FCV}(A)}{\lambda\alpha{::}K.\, A\, \alpha \rightarrow A'} \qquad \text{(R-ETA)}$$

$$\frac{A \rightarrow \texttt{<=}A'{::}K'\texttt{>}}{x\rho_{A}! \rightarrow A'} \qquad \text{(R-ABBREV)}$$

These rules constitute a *parallel* rewriting relation. A parallel rewriting relation is one in which a single reduction step on A can rewrite all of A's sub-constructors by one reduction step in parallel. For example, the R-BETA rule allows A_1 and A_2 to be rewritten in parallel with the β-reduction. Non-parallel rewriting relations often require multiple steps to accomplish the same reduction (e.g., one step for each sub-component followed by one step for the β-reduction).

Being able to do parallel reductions in a single step is crucial to proving the diamond property. For example, if $(\lambda\alpha{::}K.\, A_1)\, A_2$ rewrites in one step to $(\lambda\alpha{::}K.\, A_1)\, A_2'$ (via $A_2 \rightarrow A_2'$) and to $[A_2/\alpha]A_1$, we need to be able to show that $[A_2/\alpha]A_1 \rightarrow [A_2'/\alpha]A_1$. This reduction requires in a single step that we be able to rewrite in parallel A_2 to A_2' at each of the points in A_1 where α occurs. Only a parallel rewriting relation can provide this ability.

The R-EXT and R-ABBREV rules handle rewriting constructor extractions. The R-EXT rule handles both rewriting the tag and moving it to the right in the path. The existence of the tag allows $x\rho$'s type to be computed by a series of small steps (e.g., R-EXT only reduces the tag a single step at a time), resulting in a simple rewriting relation.

R-EXT permits moving the tag an arbitrary amount (ρ') to the right in a single step, so long as the tag has the right form (i.e., $\mathcal{S}(A, x\rho, \rho')$ exists). In retrospect, I think it would have been better to only allow moving the tag by a single $.i$ at a time; the ability to move multiple steps to the right at once only makes the system more complicated.

In order to extract a constructor then, we repeatedly use R-EXT to reduce the tag; whenever the tag is a sum type, we also move the tag to the right. Eventually, if the constructor extraction is valid, we will end up with $x\rho_{<K>}!$ or $x\rho_{<=A::K>}!$ for some K and A. The first constructor is in normal form and represents the opaque case. The second constructor represents the transparent case; it can be reduced using R-ABBREV to A in a single step.

It will sometimes be useful to consider just the $\beta\eta$ or extraction (denoted by γ) components of the rewriting relation:

Definition [Tagged] 7.3.2 ($\beta\eta$-reduction)
$A_1 \rightarrow_{\beta\eta} A_2$ iff $A_1 \rightarrow A_2$ can be derived without ever using the R-ABBREV rule or the R-EXT rule with $\rho' \neq \epsilon$.

Definition [Tagged] 7.3.3 (γ-reduction)
$A_1 \rightarrow_{\gamma} A_2$ iff $A_1 \rightarrow A_2$ can be derived without ever using the R-BETA or R-ETA rules.

Here I show that equal constructors can be converted to each other. This result is the first part of proving that the rewriting relation implements equality. The second part (valid constructors that have a common reduct are equal) requires confluence and will be proved later.

Theorem [Tagged] 7.3.4 *If $\Gamma \vdash A = A' :: K$ then $\exists A_0, A_1, \ldots, A_n$, $n \geq 0$, such that:*

1. $A = A_0$
2. $A' = A_n$
3. $A_i \rightleftharpoons A_{i+1}$ *for* $0 \leq i < n$

Proof: By structural induction on the derivation of $\Gamma \vdash A = A' :: K$. Example cases:

SYM: Apply the induction hypothesis. Since bi-directional rewriting is symmetric, reversing the sequence of types obtained from the induction hypothesis then yields the desired sequence.

TRAN: Apply the induction hypothesis twice then place the two resulting sequences side by side.

DFUN: Apply the induction hypothesis twice getting sequences $A'_0, \ldots, A'_m$ from $, \vdash A'_0 = A'_m :: \Omega$ and $A''_0, \ldots, A''_r$ from $, , x{:}A'_0 \vdash A''_0 = A''_r :: \Omega$. The desired sequence is then $\Pi x{:}A'_0.\, A''_0, \ldots, \Pi x{:}A'_0.\, A''_r, \Pi x{:}A'_1.\, A''_r, \ldots, \Pi x{:}A'_m.\, A''_r$. R-DFUN and R-REFL are used to connect them with bi-directional rewriting.

BETA: Let $n = 1$. $A_0 \to A_1$ via R-BETA and R-REFL.

EXT2: Applying the induction hypothesis gives the sequence $A'_0, \ldots, A'_m$ from $, \vdash A_1 = A :: \Omega$. The desired sequence is then $x\rho_{A'_0}\rho'\rho''!, \ldots, x\rho_{A'_m}\rho'\rho'!, x\rho\rho'_{A_2}\rho''!$. R-EXT plus R-REFL are used to connect them with bi-directional rewriting.

ABBREV2: Applying the induction hypothesis gives the sequence $A'_0, \ldots, A'_m$ from $, \vdash A = \texttt{<=}A'{::}K'\texttt{>} :: \Omega$. The desired sequence is then $x\rho_{A'_0}!, \ldots, x\rho_{A'_m}!, A'$. R-EXT plus R-ABBREV and R-REFL are used to connect them with bi-directional rewriting.

□

Corollary [Tagged] 7.3.5 (Partial correctness I)
If $, \vdash A = A' :: K$ *then* $A \approx A'$.

Proof: Apply Theorem 7.3.4 then use the fact that conversion is the transitive closure of bi-directional rewriting. □

The rest of this section is devoted to proving the basic properties I shall need about the rewriting relation. First I show that rewriting never introduces new variables; this result is needed to prove the diamond property.

Theorem [Tagged] 7.3.6 (Reduct variables) *If* $A \to A'$ *then* $FCV(A') \subseteq FCV(A)$ *and* $FTV(A') \subseteq FTV(A)$.

Proof: By structural induction on the derivation of $A \to A'$, using Lemmas 6.8.6 and 6.8.4. □

Second, I show that except for constructor functions, applications, and extractions, rewriting preserves the shape of constructors and acts in a component-wise manner. This result, when combined with confluence and the fact that the rewriting relation implements equality, will establish the needed properties of equality (e.g., equality acts in a component-wise manner).

Lemma [Tagged] 7.3.7 (Shape preservation)

1. *If* $A \to A'$ *and either* $A = \alpha$, $A = \textbf{rec}$, $A = \textbf{ref}$, $A = {<}K{>}$, *or* $A = x\rho_{<K>}!$ *then* $A = A'$.
2. *If* $\Sigma x{:}A_1.\, A_2 \to A'$ *then* $\exists A_1', A_2'$ *such that* $A' = \Sigma x{:}A_1'.\, A_2'$, $A_1 \to A_1'$, *and* $A_2 \to A_2'$.
3. *If* $\Pi x{:}A_1.\, A_2 \to A'$ *then* $\exists A_1', A_2'$ *such that* $A' = \Pi x{:}A_1'.\, A_2'$, $A_1 \to A_1'$, *and* $A_2 \to A_2'$.
4. *If* ${<}{=}A_1{::}K{>} \to A'$ *then* $\exists A_1'$ *such that* $A' = {<}{=}A_1'{::}K{>}$, *and* $A_1 \to A_1'$.
5. *If* $\textbf{ref}\, A_1 \to A'$ *then* $\exists A_1'$ *such that* $A' = \textbf{ref}\, A_1'$, *and* $A_1 \to A_1'$.
6. *If* $\textbf{rec}\, A_1 \to A'$ *then* $\exists A_1'$ *such that* $A' = \textbf{rec}\, A_1'$, *and* $A_1 \to A_1'$.

Proof: Proved sequentially (the case for ${<}K{>}$ is needed to prove the case for $x\rho_{<K>}!$) by casing on the possible rewriting rules used and using R-REFL as needed. □

Finally, I establish the consequences of having a parallel rewriting relation for use in proving the diamond property:

Lemma [Tagged] 7.3.8 *If* $A_2 \to A_2'$ *then* $[A_2/\alpha]A_1 \to [A_2'/\alpha]A_1$.

Lemma [Tagged] 7.3.9 (Substitution for non-free variables)

1. *If* $\alpha \notin FCV(A)$ *then* $[A'/\alpha]A = A$.
2. *If* $x' \notin FTV(A)$ *then* $[x\rho/x']A = A$.
3. *If* $\alpha \notin FCV(,)$ *then* $[A'/\alpha], \ = , $.
4. *If* $x' \notin FTV(,)$ *then* $[x\rho/x'], \ = , $.

Proof: Proved sequentially using structural induction on A and , . □

Lemma [Tagged] 7.3.10 (Iterated substitutions I)

1. *If* $\alpha \neq \alpha'$ *and* $\alpha' \notin FCV(A)$ *then*

$$[A/\alpha]([A_1/\alpha']A_2) = [[A/\alpha]A_1/\alpha']([A/\alpha]A_2)$$

2. $[x\rho/x']([A/\alpha]A') = [[x\rho/x']A/\alpha]([x\rho/x']A')$

3. *If* $\mathcal{S}(A', x\rho, \rho')$ *exists then*

$$[A/\alpha]\mathcal{S}(A', x\rho, \rho') = \mathcal{S}([A/\alpha]A', x\rho, \rho')$$

Proof: Proved sequentially using structural induction on A_2, A', and ρ' respectively using Lemmas 7.3.9, 6.8.6, and 6.8.4 as needed. □

Theorem [Tagged] 7.3.11 (Parallelism I) *If* $A_1 \to A_1'$ *and* $A_2 \to A_2'$ *then* $[A_2/\alpha]A_1 \to [A_2'/\alpha]A_1'$.

Proof: By structural induction on the derivation of $A_1 \to A_1'$ using Lemmas 6.8.6 and 7.3.6 as needed. Lemma 7.3.8 handles the R-REFL case; lemma 7.3.10 is needed for the R-BETA and R-EXT cases. □

Lemma [Tagged] 7.3.12 (Iterated substitutions II)

1. *If* $x_1' \neq x_2'$ *and* $x_2' \neq x_1$ *then*

$$[x_1\rho_1/x_1']([x_2\rho_2/x_2']A) = [[x_1\rho_1/x_1']x_2\rho_2/x_2']([x_1\rho_1/x_1']A)$$

2. *If* $\mathcal{S}(A, x''\rho', \rho'')$ *exists then*

$$[x\rho/x']\mathcal{S}(A, x''\rho', \rho'') = \mathcal{S}([x\rho/x']A, [x\rho/x']x''\rho', \rho'')$$

Proof: Proved sequentially by structural induction on A and ρ'' respectively. □

Theorem [Tagged] 7.3.13 (Parallelism II) *If* $A \to A'$ *then*

1. $[x\rho/x']A \to [x\rho/x']A'$

2. *If* $\mathcal{S}(A, x\rho, \rho')$ *exists then* $\mathcal{S}(A, x\rho, \rho') \to \mathcal{S}(A', x\rho, \rho')$.

Proof: Proved sequentially by structural induction on the derivation of $A \to A'$ and ρ' respectively using Lemma 6.8.6 as needed. Lemma 7.3.10 handles the R-BETA case; lemma 7.3.12 handles the R-EXT case. Lemma 7.3.7 is needed for part 2. □

7.4 Argument Kind Erasure

The erasure function which transforms constructors from the tagged system into the *erased system* is as follows:

Definition [Tagged] 7.4.1 (Argument kind erasure)

$$\begin{array}{rcl}
(\lambda\alpha{::}K.\,A)^\circ & = & \lambda\alpha.\,A^\circ \\
\\
(\Pi x{:}A_1.\,A_2)^\circ & = & \Pi x{:}A_1^\circ.\,A_2^\circ \\
(\Sigma x{:}A_1.\,A_2)^\circ & = & \Sigma x{:}A_1^\circ.\,A_2^\circ \\
(A_1\,A_2)^\circ & = & A_1^\circ\,A_2^\circ \\
{<}{=}A{::}K{>}^\circ & = & {<}{=}A^\circ{::}K{>} \\
x\rho_{A}\rho'!^\circ & = & x\rho_{A^\circ}\rho'! \\
\\
\alpha^\circ & = & \alpha \\
\mathbf{rec}^\circ & = & \mathbf{rec} \\
\mathbf{ref}^\circ & = & \mathbf{ref} \\
{<}K{>}^\circ & = & {<}K{>}
\end{array}$$

The erased system is obtained by taking the tagged system without type rules and replacing $\lambda\alpha{::}K.\,A$ in the definition of constructors with $\lambda\alpha.\,A$. The operators and rewriting relation are modified in the obvious ways:

Definition [Erased] 7.4.2 (Operator changes)

$$\begin{array}{rcll}
FCV(\lambda\alpha.\,A) & = & FCV(A) \Leftrightarrow \{\alpha\} & \\
\\
FTV(\lambda\alpha.\,A) & = & FCV(A) & \\
\\
[A/\alpha]\lambda\alpha'.\,A' & = & \lambda\alpha'.\,[A/\alpha]A' & \quad (\alpha' \neq \alpha,\ \ \alpha' \notin FCV(A)) \\
\\
[x\rho/x']\lambda\alpha.\,A & = & \lambda\alpha.\,[x\rho/x']A &
\end{array}$$

Definition [Erased] 7.4.3 (Rule changes)

$$\frac{A \rightarrow A'}{\lambda\alpha.\,A \rightarrow \lambda\alpha.\,A'} \qquad \text{(R-LAM3)}$$

$$\frac{A_1 \rightarrow A_1' \qquad A_2 \rightarrow A_2'}{(\lambda\alpha.\,A_1)\,A_2 \rightarrow [A_2'/\alpha]A_1'} \qquad \text{(R-BETA3)}$$

$$\frac{A \rightarrow A' \qquad \alpha \notin FCV(A)}{\lambda\alpha.\,A\,\alpha \rightarrow A'} \qquad \text{(R-ETA3)}$$

(These rules replace R-LAM, R-BETA, and R-ETA.)

I shall need to show many of the same properties for the rewriting relation of the erased system as I did for the tagged rewriting relation in order to prove confluence for the erased system. Rather than duplicate work, I shall relate the two systems using erasure and an *unerasure* function then transfer the needed results from the tagged system. First though, I shall need some results about how erasure interacts with the other operators:

Lemma [Tagged] 7.4.4

1. *$FCV(A^\circ) = FCV(A)$*
2. *$FTV(A^\circ) = FTV(A)$*

Lemma [Tagged] 7.4.5

1. *$([A_2/\alpha]A_1)^\circ = [A_2^\circ/\alpha](A_1^\circ)$*
2. *$([x\rho/x']A)^\circ = [x\rho/x'](A^\circ)$*
3. *If $\mathcal{S}(A, x\rho, \rho')$ exists then $\mathcal{S}(A, x\rho, \rho')^\circ = \mathcal{S}(A^\circ, x\rho, \rho')$.*

Proof: Proved sequentially by structural induction on A, A, and ρ' respectively. Examples:

$$([A/\alpha]\lambda\alpha'{::}K.\, A')^\circ = (\lambda\alpha'{::}K.\, [A/\alpha]A')^\circ = \lambda\alpha'.\, ([A/\alpha]A')^\circ$$
$$= \lambda\alpha'.\, [A^\circ/\alpha](A'^\circ) = [A^\circ/\alpha]\lambda\alpha'.\, (A'^\circ) = [A^\circ/\alpha]((\lambda\alpha'{::}K.\, A')^\circ)$$

$$([x\rho/x']\lambda\alpha'{::}K.\, A')^\circ = (\lambda\alpha'{::}K.\, [x\rho/x']A')^\circ = \lambda\alpha'.\, ([x\rho/x']A')^\circ$$
$$= \lambda\alpha'.\, [x\rho/x'](A'^\circ) = [x\rho/x']\lambda\alpha'.\, (A'^\circ) = [x\rho/x']((\lambda\alpha'{::}K.\, A')^\circ)$$

$$\mathcal{S}(\Sigma x'{:}A_1.\, A_2, x\rho, .2\rho')^\circ = \mathcal{S}([x\rho.1/x']A_2, x\rho.2, \rho')^\circ = \mathcal{S}(([x\rho.1/x']A_2)^\circ, x\rho.2, \rho')$$
$$= \mathcal{S}([x\rho.1/x'](A_2^\circ), x\rho.2, \rho') = \mathcal{S}(\Sigma x'{:}A_1^\circ.\, A_2^\circ, x\rho, .2\rho') = \mathcal{S}((\Sigma x'{:}A_1.\, A_2)^\circ, x\rho, .2\rho')$$

Where $\alpha' \neq \alpha$ and $\alpha' \notin \mathrm{FCV}(A) = \mathrm{FCV}(A^\circ)$ by Lemma 7.4.4. □

Lemma [Tagged] 7.4.6 *If $A_1 \to A_2$ then $A_1^\circ \to A_2^\circ$.*

Proof: By structural induction on the derivation of $A_1 \to A_2$. □

Corollary [Tagged] 7.4.7 *If* $A_1 \approx A_2$ *then* $A_1^\circ \approx A_2^\circ$.

Proof: Proved by structural induction on the length of the conversion using Lemma 7.4.6 to convert from tagged-system rewriting steps to erased-system rewriting steps. □

The unerasure function is defined as follows:

Definition [Erased] 7.4.8 (Argument kind unerasure)

$$\begin{array}{rcl}
\overline{\lambda\alpha.\,A} & = & \lambda\alpha{::}\Omega.\,\overline{A} \\
\\
\overline{\Pi x{:}A_1.\,A_2} & = & \Pi x{:}\overline{A_1}.\,\overline{A_2} \\
\overline{\Sigma x{:}A_1.\,A_2} & = & \Sigma x{:}\overline{A_1}.\,\overline{A_2} \\
\overline{A_1\,A_2} & = & \overline{A_1}\,\overline{A_2} \\
\overline{<=A{::}K>} & = & <=\overline{A}{::}K> \\
\overline{x\rho_A\rho'!} & = & x\rho_{\overline{A}}\rho'! \\
\\
\overline{\alpha} & = & \alpha \\
\overline{\mathbf{rec}} & = & \mathbf{rec} \\
\overline{\mathbf{ref}} & = & \mathbf{ref} \\
\overline{<K>} & = & <K>
\end{array}$$

It simply inserts Ω as the kind of each argument. This insertion will not in general produce valid constructors, but it suffices to give the needed properties; note in particular Lemma 7.4.12 below:

Lemma [Erased] 7.4.9 $(\overline{A})^\circ = A$

Corollary [Erased] 7.4.10

1. $FCV(\overline{A}) = FCV(A)$
2. $FTV(\overline{A}) = FTV(A)$

Proof: Substitute $\overline{A}$ for A in Lemma 7.4.4 then apply Lemma 7.4.9. □

Lemma [Erased] 7.4.11 *If* $\mathcal{S}(A, x\rho, \rho')$ *exists then* $\mathcal{S}(\overline{A}, x\rho, \rho')$ *exists.*

Lemma [Erased] 7.4.12 *If* $A_1 \rightarrow A_2$ *then* $\overline{A_1} \rightarrow \overline{A_2}$.

Proof: By structural induction on the derivation of $A_1 \to A_2$. □

Using these results about erasure and unerasure, the needed theorems and lemmas can be transfered from the tagged to the erased system:

Lemma [Erased] 7.4.13 *If $\mathcal{S}(A, x\rho_1, \rho_2\rho_3)$ exists then*

$$\mathcal{S}(A, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(A, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3)$$

Proof:

$$\begin{array}{lll} & \mathcal{S}(A, x\rho_1, \rho_2\rho_3) \text{ exists} & \text{(given)} \\ \Rightarrow & \mathcal{S}(\overline{A}, x\rho_1, \rho_2\rho_3) \text{ exists} & \text{(Lemma 7.4.11)} \\ \Rightarrow & \mathcal{S}(\overline{A}, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(\overline{A}, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3) & \text{(Lemma 6.8.1)} \\ \Rightarrow & \mathcal{S}(\overline{A}, x\rho_1, \rho_2\rho_3)^\circ = \mathcal{S}(\mathcal{S}(\overline{A}, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3)^\circ & \text{(erase both sides)} \\ \Rightarrow & \mathcal{S}(\overline{A}^\circ, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(\overline{A}^\circ, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3) & \text{(Lemma 7.4.5)} \\ \Rightarrow & \mathcal{S}(A, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(A, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3) & \text{(Lemma 7.4.9)} \end{array}$$

□

Theorem [Erased] 7.4.14 (Reduct variables) *If $A_1 \to A_2$ then $FCV(A_2) \subseteq FCV(A_1)$ and $FTV(A_2) \subseteq FTV(A_1)$.*

Proof: The constructor variable case is as follows:

$$\begin{array}{lll} & A_1 \to A_2 & \text{(given)} \\ \Rightarrow & \overline{A_1} \to \overline{A_2} & \text{(Lemma 7.4.12)} \\ \Rightarrow & \text{FCV}(\overline{A_2}) \subseteq \text{FCV}(\overline{A_1}) & \text{(Theorem 7.3.6)} \\ \Rightarrow & \text{FCV}(A_2) \subseteq \text{FCV}(A_1) & \text{(Corollary 7.4.10)} \end{array}$$

The term variable case is similar. □

Lemma [Erased] 7.4.15 (Shape preservation)

1. *$\alpha \to A$ implies that $A = \alpha$.*
2. *$<K> \to A$ implies that $A = <K>$.*
3. *$<=A_1::K> \to A$ implies that A has form $<=A_2::K>$ for some A_2 where $A_1 \to A_2$.*

Proof: All are proved the same way. Example case:

$$\begin{array}{lll}
 & \texttt{<=}A_1\texttt{::}K\texttt{>} \to A & \text{(given)} \\
\Rightarrow & \texttt{<=}\overline{A_1}\texttt{::}K\texttt{>} \to \overline{A} & \text{(Lemma 7.4.12)} \\
\Rightarrow & \exists A_3 \text{ such that } \overline{A} = \texttt{<=}A_3\texttt{::}K\texttt{>} & \text{(Lemma 7.3.7)} \\
\Rightarrow & (\overline{A})^\circ = \texttt{<=}A_3^\circ\texttt{::}K\texttt{>} & \text{(erase both sides)} \\
\Rightarrow & A = \texttt{<=}A_3^\circ\texttt{::}K\texttt{>} & \text{(Lemma 7.4.9)}
\end{array}$$

□

Lemma [Erased] 7.4.16 (Substitution for non-free variables)

1. *If $\alpha \notin FCV(A)$ then $[A'/\alpha]A = A$.*
2. *If $x' \notin FTV(A)$ then $[x\rho/x']A = A$.*

Proof: All are proved the same way. Example case:

$$\begin{array}{lll}
 & \alpha \notin \mathrm{FCV}(A) & \text{(given)} \\
\Rightarrow & \alpha \notin \mathrm{FCV}(\overline{A}) & \text{(Corollary 7.4.10)} \\
\Rightarrow & [\overline{A'}/\alpha]\overline{A} = \overline{A}. & \text{(Lemma 7.3.9)} \\
\Rightarrow & ([\overline{A'}/\alpha]\overline{A})^\circ = \overline{A}^\circ. & \text{(erase both sides)} \\
\Rightarrow & [\overline{A'}^\circ/\alpha]\overline{A}^\circ = \overline{A}^\circ. & \text{(Lemma 7.4.5)} \\
\Rightarrow & [A'/\alpha]A = A. & \text{(Lemma 7.4.9)}
\end{array}$$

□

Theorem [Erased] 7.4.17 (Parallelism I and II) *If $A_1 \to A_1'$ and $A_2 \to A_2'$ then*

1. $[A_2/\alpha]A_1 \to [A_2'/\alpha]A_1'$
2. $[x\rho/x']A_1 \to [x\rho/x']A_1'$
3. *If $\mathcal{S}(A_1, x\rho, \rho')$ exists then $\mathcal{S}(A_1, x\rho, \rho') \to \mathcal{S}(A_1', x\rho, \rho')$.*

Proof: All are proved the same way. Example case:

$$\begin{array}{lll}
 & A_1 \to A_1' \text{ and } A_2 \to A_2' & \text{(given)} \\
\Rightarrow & \overline{A_1} \to \overline{A_1'} \text{ and } \overline{A_2} \to \overline{A_2'} & \text{(Lemma 7.4.12)} \\
\Rightarrow & [\overline{A_2}/\alpha]\overline{A_1} \to [\overline{A_2'}/\alpha]\overline{A_1'} & \text{(Theorem 7.3.11)} \\
\Rightarrow & ([\overline{A_2}/\alpha]\overline{A_1})^\circ \to ([\overline{A_2'}/\alpha]\overline{A_1'})^\circ & \text{(Lemma 7.4.6)} \\
\Rightarrow & [\overline{A_2}^\circ/\alpha](\overline{A_1}^\circ) \to [\overline{A_2'}^\circ/\alpha](\overline{A_1'}^\circ) & \text{(Lemma 7.4.5)} \\
\Rightarrow & [A_2/\alpha]A_1 \to [A_2'/\alpha]A_1' & \text{(Lemma 7.4.9)}
\end{array}$$

□

7.5 Kind Preservation

In this section I prove confluence for the erased system by first proving the diamond property. I then use this result to show that conversion in the erased system preserves the kinds of reified constructors and whether or not reified constructors' types are transparent or opaque. This kind preservation result is required to prove subject reduction; see the next section for details.

Theorem [Erased] 7.5.1 (Diamond property)
If $A \rightarrow A_1$ and $A \rightarrow A_2$ then $\exists A_3$ such that $A_1 \rightarrow A_3$ and $A_2 \rightarrow A_3$.

Proof: By structural induction on A with case analysis on what rewriting rules were used. Without loss of generality (WLOG), we may assume that the derivation for $A \rightarrow A_1$ uses the same or an earlier rule (earlier here being defined as occuring first in definition 7.3.1) than the $A \rightarrow A_2$ derivation.

An extension of the diagram notation used in defining the diamond property (Definition 7.2.1) is used to present the subproofs. Rule names next to an arrow indicate that that rewriting step used that particular rule. The symbols Δ and Ψ represent arbitrary reduction steps. Multiple occurrences of one of these symbols in the same subproof represent the same reduction step. Solid arrows represent givens while dotted arrows are constructed according to the symbol attached to them. (IH) stands for a use of the induction hypothesis and (*lemma name*) stands for an application of the named lemma.

The forms of the constructors in the diagrams are determined by the requirements of the rules being considered and the shape preservation lemma (Lemma 7.4.15). The cases are as follows:

1. REFL vs. any rule

$$\begin{tikzcd}
& A \arrow[dl, "\mathrm{REFL}"'] \arrow[dr, "\Delta"] & \\
A \arrow[dr, dotted, "\Delta"'] & & A_2 \arrow[dl, dotted, "\mathrm{REFL}"] \\
& A_2 &
\end{tikzcd}$$

2. DFUN vs. DFUN

$$\begin{tikzcd}
 & \Pi x{:}A.\,A' \arrow[dl, "\text{DFUN}"'] \arrow[dr, "\text{DFUN}"] & \\
\Pi x{:}A_1.\,A'_1 \arrow[dr, dotted, "\text{DFUN}"'] & & \Pi x{:}A_2.\,A'_2 \arrow[dl, dotted, "\text{DFUN}"] \\
 & \Pi x{:}A_3.\,A'_3 &
\end{tikzcd}$$

Via:

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & & & A' \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\text{(IH)}"] & A'_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A'_2 \arrow[dl, dotted, "\text{(IH)}"] \\
 & A_3 & & & A'_3 &
\end{tikzcd}$$

3. DSUM vs. DSUM

$$\begin{tikzcd}
 & \Sigma x{:}A.\,A' \arrow[dl, "\text{DSUM}"'] \arrow[dr, "\text{DSUM}"] & \\
\Sigma x{:}A_1.\,A'_1 \arrow[dr, dotted, "\text{DSUM}"'] & & \Sigma x{:}A_2.\,A'_2 \arrow[dl, dotted, "\text{DSUM}"] \\
 & \Sigma x{:}A_3.\,A'_3 &
\end{tikzcd}$$

Via:

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & & & A' \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\text{(IH)}"] & A'_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A'_2 \arrow[dl, dotted, "\text{(IH)}"] \\
 & A_3 & & & A'_3 &
\end{tikzcd}$$

4. LAM3 vs. LAM3

$$\begin{tikzcd}
 & \lambda\alpha.\,A \arrow[dl, "\mathrm{LAM3}"'] \arrow[dr, "\mathrm{LAM3}"] & \\
\lambda\alpha.\,A_1 \arrow[dr, dotted, "\mathrm{LAM3}"'] & & \lambda\alpha.\,A_2 \arrow[dl, dotted, "\mathrm{LAM3}"] \\
 & \lambda\alpha.\,A_3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\text{(IH)}"] \\
 & A_3 &
\end{tikzcd}$$

5. LAM3;(REFL or APP) (R) vs. ETA3 $(\alpha \notin \mathrm{FCV}(A))$

$$\begin{tikzcd}
 & \lambda\alpha.\,A\,\alpha \arrow[dl, "\mathrm{R}"'] \arrow[dr, "\mathrm{ETA3}"] & \\
\lambda\alpha.\,A_1\,\alpha \arrow[dr, dotted, "\mathrm{ETA3}"'] & & A_2 \arrow[dl, dotted, "\Delta"] \\
 & A_3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\Delta\ \text{(IH)}"] \\
 & A_3 &
\end{tikzcd}$$

Here, $\alpha \notin \mathrm{FCV}(A_1)$ by the reduct variables lemma (Lemma 7.4.14).

6. LAM3;BETA3 (R) vs. ETA3 (WLOG $\alpha \neq \alpha'$; $\alpha \notin \mathrm{FCV}(A)$)

$$\begin{tikzcd}
 & \lambda\alpha.\,(\lambda\alpha'.\,A)\,\alpha \arrow[dl, "\mathrm{R}"'] \arrow[dr, "\mathrm{ETA3}"] & \\
\lambda\alpha.\,[\alpha/\alpha']A_1 \arrow[dr, dotted, "\Psi"'] & & A_2 \arrow[dl, dotted, "\Delta"] \\
 & A_3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \lambda\alpha'.\,A \arrow[dl, dotted, "\mathrm{LAM3}"'] \arrow[dr] & \\
\lambda\alpha'.\,A_1 \arrow[dr, dotted, "\Psi\ \text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\Delta\ \text{(IH)}"] \\
 & A_3 &
\end{tikzcd}$$

Here, $\lambda\alpha'.\,A \rightarrow \lambda\alpha'.\,A_1$ follows via the R-LAM3 rule from the given (not shown) $A \rightarrow A_1$. Since $\alpha \notin \mathrm{FCV}(A_1)$ by the reduct variables lemma, $\lambda\alpha'.\,A_1 = \lambda\alpha.\,[\alpha/\alpha']A_1$ (two constructors are considered the same if they are related by α-conversion).

7. APP vs. APP

$$\begin{tikzcd}
& A\,A' \arrow[dl, "\mathrm{APP}"'] \arrow[dr, "\mathrm{APP}"] & \\
A_1\,A_1' \arrow[dr, dotted, "\mathrm{APP}"'] & & A_2\,A_2' \arrow[dl, dotted, "\mathrm{APP}"] \\
& A_3\,A_3' &
\end{tikzcd}$$

Via:

$$\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & & & A' \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\mathrm{(IH)}"'] & & A_2 \arrow[dl, dotted, "\mathrm{(IH)}"] & A_1' \arrow[dr, dotted, "\mathrm{(IH)}"'] & & A_2' \arrow[dl, dotted, "\mathrm{(IH)}"] \\
& A_3 & & & A_3' &
\end{tikzcd}$$

8. APP;(REFL or LAM3) (R) vs. BETA3

$$\begin{tikzcd}
& (\lambda\alpha.\,A)\,A' \arrow[dl, "\mathrm{R}"'] \arrow[dr, "\mathrm{BETA3}"] & \\
(\lambda\alpha.\,A_1)\,A_1' \arrow[dr, dotted, "\mathrm{BETA3}"'] & & {[A_2'/\alpha]A_2} \arrow[dl, dotted, "\text{(parallelism I)}"] \\
& {[A_3'/\alpha]A_3} &
\end{tikzcd}$$

Via:

$$\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & & & A' \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\mathrm{(IH)}"'] & & A_2 \arrow[dl, dotted, "\mathrm{(IH)}"] & A_1' \arrow[dr, dotted, "\mathrm{(IH)}"'] & & A_2' \arrow[dl, dotted, "\mathrm{(IH)}"] \\
& A_3 & & & A_3' &
\end{tikzcd}$$

9. APP;ETA3 (R) vs. BETA3 $(\alpha \notin \mathrm{FCV}(A))$

$$\begin{tikzcd}
 & (\lambda\alpha.\, A\,\alpha)\,A' \arrow[dl, "\mathrm{R}"'] \arrow[dr, "\mathrm{BETA3}"] & \\
A_1\,A'_1 \arrow[dr, dotted, "\text{(parallelism I)}"'] & & {[A'_2/\alpha]A_2} \arrow[dl, dotted, "\text{(parallelism I)}"] \\
 & {[A'_3/\alpha]A_3} &
\end{tikzcd}$$

Via:

$$\begin{tikzcd}
 & A\,\alpha \arrow[dl, dotted, "\mathrm{APP}"'] \arrow[dr] & & & A' \arrow[dl] \arrow[dr] & \\
A_1\,\alpha \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\text{(IH)}"] & A'_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A'_2 \arrow[dl, dotted, "\text{(IH)}"] \\
 & A_3 & & & A'_3 &
\end{tikzcd}$$

Here, $A\,\alpha \to A_1\,\alpha$ follows via the R-APP and R-REFL rules from the given (not shown) $A \to A_1$. Applying parallelism I to $A'_1 \to A'_3$ and $A_1\,\alpha \to A_3$ yields $[A'_1/\alpha](A_1\,\alpha) = A_1\,A'_1 \to [A'_3/\alpha]A_3$. ($[A'_1/\alpha]A_1 = A_1$ by Lemma 7.4.16 since $\alpha \notin \mathrm{FCV}(A_1)$ by the reduct variables lemma.)

10. TRANS vs. TRANS

$$\begin{tikzcd}
 & {<=}A{::}K{>} \arrow[dl, "\mathrm{TRANS}"'] \arrow[dr, "\mathrm{TRANS}"] & & & A \arrow[dl] \arrow[dr] & \\
{<=}A_1{::}K{>} \arrow[dr, dotted, "\mathrm{TRANS}"'] & & {<=}A_2{::}K{>} \arrow[dl, dotted, "\mathrm{TRANS}"] & A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\text{(IH)}"] \\
 & {<=}A_3{::}K{>} & & & A_3 &
\end{tikzcd}$$

11. EXT vs. EXT

$$\begin{tikzcd}
& x\rho_1{}_{A}\rho_2\rho_3\rho_4! \arrow[dl, "\text{EXT}"'] \arrow[dr, "\text{EXT}"] & \\
x\rho_1\rho_2{}_{A_1'}\rho_3\rho_4! \arrow[dr, dotted, "\text{EXT}"'] & & x\rho_1\rho_2\rho_3{}_{A_2''}\rho_4! \arrow[dl, dotted, "\text{EXT}"] \\
& x\rho_1\rho_2\rho_3{}_{A_3''}\rho_4! &
\end{tikzcd}
\qquad
\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & A_2 \arrow[dl, dotted, "\text{(IH)}"] \\
& A_3 &
\end{tikzcd}$$

Where $A_1' = \mathcal{S}(A_1, x\rho_1, \rho_2)$ and $A_2'' = \mathcal{S}(A_2, x\rho_1, \rho_2\rho_3)$. By parallelism II, we have that $A_3' = \mathcal{S}(A_3, x\rho_1, \rho_2)$ and $A_3'' = \mathcal{S}(A_3, x\rho_1, \rho_2\rho_3)$ exist with $A_1' \to A_3'$ and $A_2'' \to A_3''$. By Lemma 7.4.13,

$$A_3'' = \mathcal{S}(A_3, x\rho_1, \rho_2\rho_3) = \mathcal{S}(\mathcal{S}(A_3, x\rho_1, \rho_2), x\rho_1\rho_2, \rho_3) = \mathcal{S}(A_3', x\rho_1\rho_2, \rho_3)$$

12. EXT vs. ABBREV

$$\begin{tikzcd}
& x\rho{}_{A}! \arrow[dl, "\text{EXT}"'] \arrow[dr, "\text{ABBREV}"] & \\
x\rho{}_{A_1}! \arrow[dr, dotted, "\text{ABBREV}"'] & & A_2 \arrow[dl, dotted, "\text{(shape)}"] \\
& A_3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\text{(IH)}"'] & & <\!=\!A_2::K\!> \arrow[dl, dotted, "\text{(IH)}"] \\
& <\!=\!A_3::K\!> &
\end{tikzcd}$$

13. BETA3 vs. BETA3

$$\begin{tikzcd}
& (\lambda\alpha.\, A)\, A' \arrow[dl, "\text{BETA3}"'] \arrow[dr, "\text{BETA3}"] & \\
{[A_1'/\alpha]A_1} \arrow[dr, dotted, "\text{(parallelism I)}"'] & & {[A_2'/\alpha]A_2} \arrow[dl, dotted, "\text{(parallelism I)}"] \\
& {[A_3'/\alpha]A_3} &
\end{tikzcd}$$

Via:

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & & & A' \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "(\mathrm{IH})"'] & & A_2 \arrow[dl, dotted, "(\mathrm{IH})"] & A'_1 \arrow[dr, dotted, "(\mathrm{IH})"'] & & A'_2 \arrow[dl, dotted, "(\mathrm{IH})"] \\
 & A_3 & & & A'_3 &
\end{tikzcd}$$

14. ETA3 vs. ETA3 $(\alpha \notin \mathrm{FCV}(A))$

$$\begin{tikzcd}
 & \lambda\alpha.\, A\,\alpha \arrow[dl, "\mathrm{ETA3}"'] \arrow[dr, "\mathrm{ETA3}"] & & & A \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "\Psi"'] & & A_2 \arrow[dl, dotted, "\Delta"] & A_1 \arrow[dr, dotted, "\Psi\ (\mathrm{IH})"'] & & A_2 \arrow[dl, dotted, "\Delta\ (\mathrm{IH})"] \\
 & A_3 & & & A_3 &
\end{tikzcd}$$

15. ABBREV vs. ABBREV

$$\begin{tikzcd}
 & {}^{x\rho}A^{!} \arrow[dl, "\mathrm{ABBREV}"'] \arrow[dr, "\mathrm{ABBREV}"] & & & A \arrow[dl] \arrow[dr] & \\
A_1 \arrow[dr, dotted, "(\mathrm{shape})"'] & & A_2 \arrow[dl, dotted, "(\mathrm{shape})"] & {<=}A_1{::}K{>} \arrow[dr, dotted, "(\mathrm{IH})"'] & & {<=}A_2{::}K{>} \arrow[dl, dotted, "(\mathrm{IH})"] \\
 & A_3 & & & {<=}A_3{::}K{>} &
\end{tikzcd}$$

□

Theorem [Erased] 7.5.2 (Confluence)
If $A \rightarrow^* A_1$ and $A \rightarrow^* A_2$ then $\exists A'$ such that $A_1 \rightarrow^* A'$ and $A_2 \rightarrow^* A'$.

Proof: Same as that of Theorem 7.2.2. □

Theorem [Erased] 7.5.3 (Church-Rosser)
If $A_1 \approx A_2$ then $A_1 \downarrow^* A_2$.

Proof: (Idea) $A_1 \approx A_2$ means that one can get from A_1 to A_2 via a finite alternating series of first rewriting then rewriting backwards. Without loss of generality, we can assume the series starts with rewriting and ends with rewriting. (This is because we can always rewrite (backwards) a term to itself via the R-REFL rule.) We can then use confluence to "fill in the bottom half of the gird" like we did for the previous theorem. Example:

$$\begin{tikzcd}
A_1 \arrow[dr, "*"] & & A_6 \arrow[dl, "*"'] \arrow[dr, "*"] & & A_5 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
& A_7 \arrow[dr, dotted, "*"] & & A_8 \arrow[dl, dotted, "*"'] \arrow[dr, dotted, "*"] & & A_2 \arrow[dl, dotted, "*"'] \\
& & A_9 \arrow[dr, dotted, "*"] & & A_{10} \arrow[dl, dotted, "*"'] & \\
& & & A' & &
\end{tikzcd}$$

The result we want follows by the transitivity of $\rightarrow^*$. □

Lemma [Erased] 7.5.4 (Kind Preservation)

1. *If $<=A::K> \approx <=A'::K'>$ then $K = K'$.*
2. *$<=A::K> \approx <K'>$ is impossible.*

Proof: Applying Theorem 7.5.3 shows that the two sides of each conversion have a common reduct. Repeated application of Lemma 7.4.15 places requirements on the form the common reduct can have. In the first case, the common reduct must be a transparent type whose kind is the same as both K and K'. Hence, K must equal K'. In the second case, the conversion is impossible because it would require the common reduct to be both a transparent and an opaque type at the same time. □

7.6 Subject Reduction

In this section I prove subject reduction for constructors. Because I have not yet shown that all sub-constructors of a constructor are valid (in particular, tags are not yet shown to be valid), I have to do this via a series of steps. First I prove the propositions needed to show that β- or η-reducing an entire valid constructor with no reduction of any sub-constructors results in a valid constructor:

Lemma [Tagged] 7.6.1 (Substitutions on assignments)
Let θ be a place or constructor substitution. Then:

1. *$dom(\theta, \ ) = dom(, \ )$*
2. *$\theta(, _1; , _2) = (\theta, _1); (\theta, _2)$*
3. *If $\alpha{::}K \in \ ,$ then $\alpha{::}K \in \theta, \ .$*
4. *If $x{:}A \in \ ,$ then $x{:}(\theta A) \in \theta, \ .$*

Proof: The first two parts are shown by structural induction on , and , $_2$ respectively. For the last two parts, first show by structural induction that:

$$\theta(\bullet, D_1, \ldots, D_n) = \theta\bullet, \theta D_1, \ldots, \theta D_n$$

The desired results then follow from this plus the facts that $\theta(\alpha{::}K) = \alpha{::}K$ and $\theta(x{:}A) = x{:}(\theta A)$. □

Theorem [Tagged] 7.6.2 (Validity of constructor substitution)
Suppose , $\vdash A :: K$. Then:

1. *If $\vdash \ , \ ; \alpha{::}K; \ ,'$ valid then $\vdash \ , \ ; [A/\alpha], \ '$ valid.*
2. *If $, \ ; \alpha{::}K; \ , ' \vdash A' :: K'$ then $, \ ; [A/\alpha], \ ' \vdash [A/\alpha]A' :: K'$.*
3. *If $, \ ; \alpha{::}K; \ , ' \vdash x\rho \Rightarrow A'$ then $, \ ; [A/\alpha], \ ' \vdash x\rho \Rightarrow [A/\alpha]A'$.*
4. *If $, \ ; \alpha{::}K; \ , ' \vdash A_1 = A_2 :: K'$ then $, \ ; [A/\alpha], \ ' \vdash [A/\alpha]A_1 = [A/\alpha]A_2 :: K'$.*

Proof: By simultaneous structural induction on the derivations. Example cases:

DECL-T: Given $\vdash \ , \ , \alpha{::}K; \ , '', x{:}A'$ valid derived via rule DECL-T from $, \ , \alpha{::}K; \ , '' \vdash A' :: \Omega$ and $x \notin$ dom$(, \ , \alpha{::}K; \ , '')$. By the induction hypothesis, we have that $, \ ; [A/\alpha], \ '' \vdash [A/\alpha]A' :: \Omega$. By Lemma 6.8.2, dom$(, \ , \alpha{::}K; \ , '')$ = dom$(, \ ) \cup \{\alpha\} \cup$dom$(, '')$ so we have that $x \notin$ dom$(, \ )$ and $x \notin$ dom$(, '')$. Since by Lemma 7.6.1, dom$([A/\alpha], '')$ = dom$(, '')$, we have that $x \notin$ dom$(, \ ; [A/\alpha], '')$. Hence, by rule DECL-T, we have that $\vdash \ , \ ; [A/\alpha], '', x{:}A'$ valid.

C-VAR I: Given $, ; \alpha{::}K; , ' \vdash \alpha :: K'$ derived via rule C-VAR from $\vdash , ; \alpha{::}K; , '$ valid and $\alpha{::}K' \in , ; \alpha{::}K; , '$. By Lemma 6.8.11, $K = K'$. By the induction hypothesis, we have that $\vdash , ; [A/\alpha], '$ valid. By weakening (Theorem 6.8.10), we have that $, ; [A/\alpha], ' \vdash A :: K \Rightarrow , ; [A/\alpha], ' \vdash [A/\alpha]\alpha :: K'$.

C-VAR II: Given $, , \alpha{::}K; , ' \vdash \alpha' :: K'$, $\alpha \neq \alpha'$, derived via rule C-VAR from $\vdash , , \alpha{::}K; , '$ valid and $\alpha'{::}K' \in , ; \alpha{::}K; , '$. By Lemma 7.6.1, $\alpha'{::}K' \in , ; [A/\alpha], '$. By the induction hypothesis, we have that $\vdash , ; [A/\alpha], '$ valid. Hence, by rule C-VAR and the fact that $[A/\alpha]\alpha' = \alpha'$,
$, ; [A/\alpha], ' \vdash [A/\alpha]\alpha' :: K'$.

P-INIT: Given $, , \alpha{::}K; , ' \vdash x \Rightarrow A'$ derived via rule P-INIT from $\vdash , , \alpha{::}K; , '$ valid and $x{:}A' \in , , \alpha{::}K; , '$. Applying the induction hypothesis gives us that $\vdash , ; [A/\alpha], '$ valid. By Lemma 7.6.1, $x{:}[A/\alpha]A' \in [A/\alpha], ; [A/\alpha], '$. By repeated use of Lemma 6.8.3, we have that $\vdash , , \alpha{::}K$ valid $\Rightarrow \alpha \notin \mathrm{FCV}(,)$ (Theorem 6.8.8) $\Rightarrow [A/\alpha], \ = \ ,$ (Lemma 7.3.9) $\Rightarrow x{:}[A/\alpha]A' \in , ; [A/\alpha], '$. From these results, the desired result follows via the P-INIT rule.

P-MOVE: Given $, , \alpha{::}K; , ' \vdash x\rho\rho' \Rightarrow A_3$ derived via rule P-MOVE from $, , \alpha{::}K; , ' \vdash x\rho \Rightarrow A_1$, $, , \alpha{::}K; , ' \vdash A_1 = A_2 :: \Omega$, and $A_3 = \mathcal{S}(A_2, x\rho, \rho')$. Applying the induction hypothesis gives us that $, ; [A/\alpha], ' \vdash x\rho \Rightarrow [A/\alpha]A_1$ and $, ; [A/\alpha], ' \vdash [A/\alpha]A_1 = [A/\alpha]A_2 :: \Omega$. By Lemma 7.3.10, $[A/\alpha]A_3 = [A/\alpha]\mathcal{S}(A_2, x\rho, \rho') = \mathcal{S}([A/\alpha]A_2, x\rho, \rho')$. From these results, the desired result follows via the P-MOVE rule.

E-ETA: Given $, , \alpha{::}K; , ' \vdash \lambda\alpha'{::}K'.\, A'\,\alpha' = A' :: K'{\Rightarrow}K''$ derived via rule E-ETA from $, , \alpha{::}K; , ' \vdash A' :: K'{\Rightarrow}K''$ and $\alpha' \notin \mathrm{FCV}(A')$. WLOG, assume $\alpha' \neq \alpha$ and $\alpha' \notin \mathrm{FCV}(A)$. By the induction hypothesis, we have that $, ; [A/\alpha], ' \vdash [A/\alpha]A' :: K'{\Rightarrow}K''$. By Lemma 6.8.6, $\mathrm{FCV}([A/\alpha]A') \subseteq \mathrm{FCV}(A) \cup (\mathrm{FCV}(A') \Leftrightarrow \{\alpha\}) \Rightarrow \alpha' \notin \mathrm{FCV}([A/\alpha]A')$. The desired result follows via the E-ETA rule.

□

Lemma [Tagged] 7.6.3 (Kind inhabitation) *If* $\vdash ,$ *valid and* K *is any kind then* $\exists A$ *such that* $, \vdash A :: K$.

Proof: Define $\overline{K}$ as:

$$\begin{array}{rcl} \overline{\Omega} & = & <\Omega> \\ \overline{K_1{\Rightarrow}K_2} & = & \lambda\alpha{::}K_1.\, \overline{K_2} \end{array}$$

Then show by structural induction on K that $, \vdash \overline{K} :: K$. □

Corollary [Tagged] 7.6.4 (Strengthening)
If $, ,\alpha{::}K \vdash A :: K'$ *and* $\alpha \notin FCV(A)$ *then* $, \vdash A :: K'$.

Proof: By Lemma 6.8.3, $\vdash ,$ valid. Apply Lemma 7.6.3 to K to get $\overline{K}$ such that $, \vdash \overline{K} :: K$. Then apply Theorem 7.6.2 with $, \; = \; , , ,' = \bullet$, $A = \overline{K}$, $K = K$, and $K' = K'$ to get $, \vdash [\overline{K}/\alpha]A :: K'$. Since $\alpha \notin$ FCV(A), we have that $[\overline{K}/\alpha]A = A$ by Lemma 7.3.9. This gives us the desired result. □

Second, I use these results and the previous section's kind preservation result to prove that equal constructors are valid constructors. (This result is needed to show that place lookup produces valid constructors and hence tags of valid constructors are valid.)

Lemma [Tagged] 7.6.5 *If* $, \vdash x\rho_A\rho'! :: K$ *then*

1. $, \vdash x\rho \Rightarrow A$ *is a sub-derivation.*
2. $, \vdash x\rho\rho' \Rightarrow A'$ *where* A' *has either the form* $<K>$ *or the form* $<=A''{::}K>$ *for some* A''

Proof: By inspection of the only two rules (C-EXT-O2 and C-EXT-T2) capable of deriving $, \vdash x\rho_A\rho'! :: K$. □

Lemma [Mixed] 7.6.6 *If* $, \vdash x\rho \Rightarrow A$, *where* $\rho = .i_1.\cdots.i_n$, $(n \geq 0)$ *then* $\exists B_0, B_0', B_1, B_1', \ldots, B_n$ *such that:*

1. $B_0 = \,, (x)^\circ$
2. $B_j' \approx B_j$ $\quad (0 \leq j < n)$
3. $B_{j+1} = \mathcal{S}(B_j', x.i_1\cdots.i_j, .i_{j+1})$ $\quad (0 \leq j < n)$
4. $B_n \approx A^\circ$

(Here A's are tagged constructors and B's are erased constructors.)

Proof: By structural induction on the derivation of $, \vdash x\rho \Rightarrow A$. The base case is handled using Theorem 6.8.11 to show that $x{:}A \in \,, \Rightarrow A = \,, (x)$ and the fact that conversion is reflexive by definition. The inductive case is handled using Corollary 7.3.5 and Corollary 7.4.7 (to turn the equality judgment into an erased conversion), Lemmas 7.4.5 and 7.4.13 (to turn multiple-step path selections into erased staged single-step selections), and the properties of conversion (to turn a possibly empty series of conversions into a single one). □

Lemma [Erased] 7.6.7 *If $A \approx A'$ and both $\mathcal{S}(A, x\rho, \rho')$ and $\mathcal{S}(A', x\rho, \rho')$ exist then $\mathcal{S}(A, x\rho, \rho') \approx \mathcal{S}(A', x\rho, \rho')$.*

Proof: By Theorem 7.5.3, A and A' have a common reduct. By repeated use of parallelism II (theorem 7.4.17), this means that $\mathcal{S}(A, x\rho, \rho')$ and $\mathcal{S}(A', x\rho, \rho')$ have a common reduct; hence they convert. □

Theorem [Tagged] 7.6.8 *If $, \vdash x\rho \Rightarrow A_1$ and $, \vdash x\rho \Rightarrow A_2$ then $A_1^\circ \approx A_2^\circ$.*

Proof: First apply Lemma 7.6.6 to both $, \vdash x\rho \Rightarrow A_1$ and $, \vdash x\rho \Rightarrow A_2$ then apply Lemma 7.6.7 as many times as needed. □

Theorem [Tagged] 7.6.9 (Replacement with an equal type)
Suppose $, \vdash A_1 = A_2 :: \Omega$ and $, \vdash A_2 :: \Omega$. Then:

1. *If $\vdash , , x{:}A_1; ,'$ valid then $\vdash , , x{:}A_2; ,'$ valid.*
2. *If $, , x{:}A_1; ,' \vdash A :: K$ then $, , x{:}A_2; ,' \vdash A :: K$.*
3. *If $, , x{:}A_1; ,' \vdash x'\rho \Rightarrow A$ then $, , x{:}A_2; ,' \vdash x'\rho \Rightarrow A$.*
4. *If $, , x{:}A_1; ,' \vdash A = A' :: K$ then $, , x{:}A_2; ,' \vdash A = A' :: K$.*

Proof: By simultaneous structural induction on the derivations. Interesting cases:

DECL-T I: Given $\vdash , , x{:}A_1$ valid derived via DECL-T from $, \vdash A_1 :: \Omega$ and $x \notin \text{dom}(,)$. The desired result, $\vdash , , x{:}A_2$ valid follows immediately from the precondition that $, \vdash A_2 :: \Omega$ via the DECL-T rule.

P-INIT I: Given $, , x{:}A_1; ,' \vdash x \Rightarrow A$ derived via rule P-INIT from $\vdash , , x{:}A_1; ,'$ valid and $x{:}A \in (, , x{:}A_1; ,')$. By applying the induction hypothesis, we have that $\vdash , , x{:}A_2; ,'$ valid. By rule P-INIT then, $, , x{:}A_2; ,' \vdash x \Rightarrow A_2$. By applying weakening to the equality precondition (theorem 6.8.10), we have that $, , x{:}A_2; ,' \vdash A_1 = A_2 :: \Omega$. Then, by applying E-SYM, we have that $, , x{:}A_2; ,' \vdash A_2 = A_1 :: \Omega$. By lemma 6.8.11, we have that $A = A_1$. Thus, by P-MOVE, we have that $, , x{:}A_2; ,' \vdash x \Rightarrow A$.

□

Theorem [Tagged] 7.6.10 (Validity of equal constructors)
If $, \vdash A_1 = A_2 :: K$ then $, \vdash A_1 :: K$ and $, \vdash A_2 :: K$.

Proof: By structural induction on the typing derivation. Example cases:

E-DFUN: Given $, \vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega$ derived via rule E-DFUN from $, \vdash A_1 = A_1' :: \Omega$ and $, , x{:}A_1 \vdash A_2 = A_2' :: \Omega$. Applying the induction hypothesis gives us that $, \vdash A_1' :: \Omega$, $, , x{:}A_1 \vdash A_2 :: \Omega$, and $, , x{:}A_1 \vdash A_2' :: \Omega$. Applying Theorem 7.6.9, we get that $, , x{:}A_1' \vdash A_2' :: \Omega$. Applying rule C-DFUN twice then gives us the desired results.

E-BETA: Given $, \vdash (\lambda\alpha{::}K.\, A)\, A' = [A'/\alpha]A :: K'$ derived via rule E-BETA from $, , \alpha{::}K \vdash A :: K'$ and $, \vdash A' :: K$. By using rules C-LAM and C-APP, we have that $, \vdash (\lambda\alpha{::}K.\, A)\, A' :: K'$. By Theorem 7.6.2, $, \vdash [A'/\alpha]A :: K'$.

E-ETA: Given $, \vdash \lambda\alpha{::}K.\, A\,\alpha = A :: K{\Rightarrow}K'$, $\alpha \notin$ FCV(,) (WLOG), derived via rule E-ETA from $, \vdash A :: K{\Rightarrow}K'$ and $\alpha \notin \mathrm{FCV}(A)$. By Lemma 6.8.3, $\vdash ,$ valid. Hence, by rules C-VAR and DECL-C, $, , \alpha{::}K \vdash \alpha :: K$. By weakening (Theorem 6.8.10), $, , \alpha{::}K \vdash A :: K{\Rightarrow}K'$. Hence, by rules C-APP and C-LAM, $, \vdash \lambda\alpha{::}K.\, A\,\alpha :: K{\Rightarrow}K'$.

E-EXT2: Given $, \vdash x\rho_{A_1}\rho'\rho''! = x\rho\rho'_{A_2}\rho''! :: K$ derived via rule E-EXT2 from $, \vdash x\rho_{A_1}\rho'\rho''! :: K$, $, \vdash A_1 = A :: \Omega$, and $\mathcal{S}(A, x\rho, \rho') = A_2$. By Lemma 7.6.5, we have that $, \vdash x\rho \Rightarrow A_1$ and $, \vdash x\rho\rho'\rho'' \Rightarrow A'$ where A' has either the form $<K>$ or the form $<=A''::K>$ for some A''. By P-MOVE then, $, \vdash x\rho\rho' \Rightarrow A_2$. Thus, by C-EXT-O2 or C-EXT-T2 (depending on the form of A''), we have that $, \vdash x\rho\rho'_{A_2}\rho''! :: K$.

E-ABBREV2: Given $, \vdash x\rho_A! = A' :: K$ derived via rule E-ABBREV2 from $, \vdash x\rho_A! :: K$ and $, \vdash A = <=A'::K'> :: \Omega$. Applying the induction hypothesis gives us that $, \vdash <=A'::K'> :: \Omega$ and hence (only rule C-TRANS can apply) that $, \vdash A' :: K'$. By Lemma 7.6.5, we have that $, \vdash x\rho \Rightarrow A$ and $, \vdash x\rho \Rightarrow A''$ where A'' has either the form $<K>$ or the form $<=A'''::K>$ for some A'''. By rule P-MOVE, $, \vdash x\rho \Rightarrow <=A'::K'>$. By Theorem 7.6.8, this implies that $<=A'^{\circ}::K'> \approx A''^{\circ}$ where A''° is either $<K>$ or $<=A'''^{\circ}::K>$. By Lemma 7.5.4, this implies that $K = K'$.

□

The following theorem, which will be needed for deciding constructor validity, also follows from the kind preservation results:

Theorem [Tagged] 7.6.11 (Uniqueness of constructor kind)
If $, \vdash A :: K_1$ *and* $, \vdash A :: K_2$ *then* $K_1 = K_2$.

Proof: By structural induction on A. Example cases:

Var: Here $A = \alpha$. The only applicable rule is C-VAR so we must have that $\alpha{::}K_1 \in$, and $\alpha{::}K_2 \in$, . By Lemma 6.8.3 and Theorem 6.8.11, $K_1 = K_2$.

Lam: Here $A = \lambda\alpha{::}K.\,A'$. The only applicable rule is C-LAM. Applying the induction hypothesis, gives us that , $,\alpha{::}K \vdash A' :: K'$ for exactly one K'. Hence, by C-LAM, $K_1 = K_2 = K{\Rightarrow}K'$.

Ext: Here $A = x\rho_{A'}\rho'!$. By Lemma 7.6.5, , $\vdash x\rho\rho' \Rightarrow A_1$ and , $\vdash x\rho\rho' \Rightarrow A_2$ where the A_i's each have either the form $<K_i>$ or the form $<=A_i'{::}K_i>$ for some A_i'. By Theorem 7.6.8, $A_1^\circ \approx A_2^\circ$. Hence, by kind preservation (Lemma 7.5.4), $K_1 = K_2$.

□

Third, I show that selection as used in place lookup results in valid constructors:

Theorem [Tagged] 7.6.12 (Validity of place substitution)
Suppose , $\vdash x\rho \Rightarrow A'$. *Then:*

1. *If* $\vdash$, $,x'{:}A';,'$ *valid then* $\vdash$, $;[x\rho/x'],'$ *valid.*

2. *If* , $,x'{:}A';,' \vdash A :: K$ *then* , $;[x\rho/x'],' \vdash [x\rho/x']A :: K$.

3. *If* , $,x'{:}A';,' \vdash x''\rho'' \Rightarrow A$ *then* , $;[x\rho/x'],' \vdash [x\rho/x']x''\rho'' \Rightarrow [x\rho/x']A$.

4. *If* , $,x'{:}A';,' \vdash A_1 = A_2 :: K$ *then* , $;[x\rho/x'],' \vdash [x\rho/x']A_1 = [x\rho/x']A_2 :: K$.

Proof: By simultaneous structural induction on the typing derivations. Example cases:

DECL-T I: Given $\vdash$, $,x'{:}A';,',x''{:}A$ valid derived via rule DECL-T from , $,x'{:}A';,' \vdash A :: \Omega$ and $x'' \notin \mathrm{dom}(,\,,x'{:}A';,') = \mathrm{dom}(,\,) \cup \{x'\} \cup \mathrm{dom}(,\,')$ by Lemma 6.8.2. By Theorem 6.8.3, $\vdash$, $,x'{:}A';,'$ valid. Applying the induction hypothesis then gives us that , $,x'{:}A';,' \vdash A :: \Omega$ and , $;[x\rho/x'],' \vdash [x\rho/x']A :: \Omega$. Since by Lemma 7.6.1, $\mathrm{dom}([x\rho/x'],\,') = \mathrm{dom}(,\,')$, we have that $x'' \notin \mathrm{dom}(,\,;[x\rho/x'],\,')$. Thus, by DECL-T, we have that $\vdash$, $;[x\rho/x'],',x''{:}[x\rho/x']A$ valid.

C-VAR: Given , $,x'{:}A';,' \vdash \alpha :: K$ derived via C-VAR from $\vdash$, $,x'{:}A';,'$ valid and $\alpha{::}K \in$ $(,\,,x'{:}A';,')$. Applying the induction hypothesis gives us that $\vdash$, $;[x\rho/x'],'$ valid. By Lemma 7.6.1, we have that $\alpha{::}K \in$, $;[x\rho/x'],'$. The desired result then follows via the C-VAR rule.

C-EXT-O2: Given $, , x'{:}A'; , ' \vdash x''\rho'{}_A\rho''! :: K$ derived via C-EXT-O2 from $, , x'{:}A'; , ' \vdash x''\rho'\rho'' \Rightarrow <\!K\!>$ and $, , x'{:}A'; , ' \vdash x''\rho' \Rightarrow A$. Applying the induction hypothesis gives us that $, ; [x\rho/x'], ' \vdash [x\rho/x']x''\rho'\rho'' \Rightarrow <\!K\!>$ and $, ; [x\rho/x'], ' \vdash [x\rho/x']x''\rho' \Rightarrow [x\rho/x']A$. The desired result then follows via the C-EXT-O2 rule since $[x\rho/x']x''\rho'{}_A\rho''! = ([x\rho/x']x''\rho')_{[x\rho/x']A}\rho''!$ and $[x\rho/x']x''\rho'\rho'' = ([x\rho/x']x''\rho')\rho''$ by Lemma 6.5.1.

P-INIT I: Given $, , x'{:}A'; , ' \vdash x' \Rightarrow A$ derived via rule P-INIT from $\vdash , , x'{:}A'; , '$ valid and $x'{:}A \in (, , x'{:}A'; , ')$. By repeated use of Lemma 6.8.3, we have that $\vdash , , x'{:}A'$ valid $\Rightarrow , \vdash A' :: \Omega$ and $x' \notin \mathrm{dom}(,)$ (DECL-T). By Theorem 6.8.5, $\mathrm{FTV}(A') \subseteq \mathrm{dom}(,)$ $\Rightarrow x' \notin \mathrm{FTV}(A') \Rightarrow [x\rho/x']A' = A'$ (Theorem 7.3.9). By Lemma 6.8.11, $A = A'$. Applying the induction hypothesis gives us that $\vdash , ; [x\rho/x'], '$ valid. By applying weakening (Theorem 6.8.10) to the precondition, we get that $, ; [x\rho/x'], ' \vdash x\rho \Rightarrow A'$. Hence, $, ; [x\rho/x'], ' \vdash [x\rho/x']x' \Rightarrow [x\rho/x']A$. as desired.

P-INIT II: Given $, , x'{:}A'; , ' \vdash x'' \Rightarrow A$, $x'' \neq x'$, derived via rule P-INIT from $\vdash , , x'{:}A'; , '$ valid and $x''{:}A \in (, , x'{:}A'; , ')$. Applying the induction hypothesis gives us that $\vdash , ; [x\rho/x'], '$ valid. By Lemma 7.6.1, $x''{:}[x\rho/x']A \in ([x\rho/x'], , x'{:}[x\rho/x']A'; [x\rho/x'], ')$. By repeated use of Lemma 6.8.3, we have that $\vdash , , x'{:}A'$ valid and $\vdash ,$ valid $\Rightarrow x' \notin \mathrm{dom}(,)$ (DECL-T). By Theorem 6.8.5, $\mathrm{FTV}(,) \subseteq \mathrm{dom}(,) \Rightarrow x' \notin \mathrm{FTV}(,) \Rightarrow [x\rho/x'], \; = ,$ (Theorem 7.3.9). Hence, $x''{:}[x\rho/x']A \in , ; [x\rho/x'], '$. The desired result then follows via the P-INIT rule.

E-BETA: By use of the induction hypothesis plus Lemma 7.3.10.

E-ETA: By use of the induction hypothesis plus Lemma 6.8.6.

E-EXT2: Given $, , x'{:}A'; , ' \vdash x''\rho'{}_{A_1}\rho''\rho'''! = x''\rho'\rho''{}_{A_2}\rho'''! :: K$ derived via rule E-EXT2 from $, , x'{:}A'; , ' \vdash x''\rho'{}_{A_1}\rho''\rho'''! :: K$, $, , x'{:}A'; , ' \vdash A_1 = A :: \Omega$, and $\mathcal{S}(A, x''\rho', \rho'') = A_2$. Applying the induction hypothesis gives us that $, ; [x\rho/x'], ' \vdash ([x\rho/x']x''\rho')_{[x\rho/x']A_1}\rho''\rho'''! :: K$ and $, ; [x\rho/x'], ' \vdash [x\rho/x']A_1 = [x\rho/x']A :: \Omega$. By Lemma 7.3.12, $[x\rho/x']A_2 = [x\rho/x']\mathcal{S}(A, x''\rho', \rho'') = \mathcal{S}([x\rho/x']A, [x\rho/x']x''\rho', \rho'')$. The desired result then follows via the E-EXT2 rule.

□

Theorem [Tagged] 7.6.13 (Validity of selection)
Suppose $, \vdash x\rho \Rightarrow A$, $, \vdash A :: \Omega$, *and* $\mathcal{S}(A, x\rho, \rho') = A'$. *Then*

1. $, \vdash A' :: \Omega$

2. $, \vdash x\rho\rho' \Rightarrow A'$

Proof: By structural induction on ρ':

Base: Given $, \vdash x\rho \Rightarrow A$, $, \vdash A :: \Omega$, and $\mathcal{S}(A, x\rho, \epsilon) = A' = A$ ($\rho' = \epsilon$). The desired results follow from this immediately.

Fst: Given $, \vdash x\rho \Rightarrow A$, $, \vdash A :: \Omega$, and $\mathcal{S}(A, x\rho, \rho'.1) = A'$. By Lemma 6.8.1, $\mathcal{S}(A, x\rho, \rho'.1) = \mathcal{S}(\mathcal{S}(A, x\rho, \rho'), x\rho\rho', .1)$. Let $A'' = \mathcal{S}(A, x\rho, \rho')$. Since $\mathcal{S}(A'', x\rho\rho', .1)$ exists and equals A', A'' must have form $\Sigma x'{:}A'.\,A_2$ for some A_2. Applying the induction hypothesis gives us that $, \vdash x\rho\rho' \Rightarrow A''$ and $, \vdash \Sigma x'{:}A'.\,A_2 :: \Omega \Rightarrow\, , \vdash A' :: \Omega$ (C-DSUM). By E-REFL, $, \vdash A'' = A'' :: \Omega$, so by P-MOVE we have $, \vdash x\rho\rho'.1 \Rightarrow A'$.

Snd: Given $, \vdash x\rho \Rightarrow A$, $, \vdash A :: \Omega$, and $\mathcal{S}(A, x\rho, \rho'.2) = A'$. By Lemma 6.8.1, $\mathcal{S}(A, x\rho, \rho'.2) = \mathcal{S}(\mathcal{S}(A, x\rho, \rho'), x\rho\rho', .2)$. Let $A'' = \mathcal{S}(A, x\rho, \rho')$. Since $\mathcal{S}(A'', x\rho\rho', .2)$ exists and equals A', A'' must have form $\Sigma x'{:}A_1.\,A_2$ for some A_1 and A_2 where $[x\rho\rho'.1/x']A_2 = A'$. Applying the induction hypothesis gives us that $, \vdash x\rho\rho' \Rightarrow A''$ and $, \vdash \Sigma x'{:}A_1.\,A_2 :: \Omega \Rightarrow\, , \vdash A_1 :: \Omega$ and $, , x'{:}A_1 \vdash A_2 :: \Omega$ (C-DSUM). By E-REFL, $, \vdash A'' = A'' :: \Omega$ and $, \vdash A_1 = A_1 :: \Omega$, so by P-MOVE we have $, \vdash x\rho\rho'.1 \Rightarrow A_1$ and $, \vdash x\rho\rho'.2 \Rightarrow A'$. By Theorem 7.6.12, we have that $, \vdash [x\rho\rho'.1/x']A_2 :: \Omega$.

□

Fourth, I show from this result and the validity of equals result that the result of looking up a place is valid and hence that tags of valid constructors are valid:

Theorem [Tagged] 7.6.14 (Validity of looked up places)
If $, \vdash x\rho \Rightarrow A$ *then* $, \vdash A :: \Omega$.

Proof: By structural induction on the typing derivation:

P-INIT: Given $, \vdash x \Rightarrow A$ derived via rule P-INIT from $\vdash ,$ valid and $x{:}A \in ,$. By Lemma 6.8.11, $, \vdash A :: \Omega$.

P-MOVE: Given $, \vdash x\rho\rho' \Rightarrow A''$ derived via rule P-MOVE from $, \vdash x\rho \Rightarrow A$, $, \vdash A = A' :: \Omega$, and $\mathcal{S}(A', x\rho, \rho') = A''$. By Theorem 7.6.10, $, \vdash A' :: \Omega$. By P-MOVE, $, \vdash x\rho \Rightarrow A'$. Hence, by Theorem 7.6.13, $, \vdash A'' :: \Omega$.

□

Theorem [Tagged] 7.6.15 (Validity of tags)
If $, \vdash x\rho_A\rho'! :: K$ *then* $, \vdash A :: \Omega$.

Proof: By Lemma 7.6.5, $, \vdash x\rho \Rightarrow A$. By Theorem 7.6.14, $, \vdash A :: \Omega$. □

Finally, I use the previous results to show subject reduction:

Theorem [Tagged] 7.6.16 (Partial correctness II)
If $, \vdash A :: K$ *and* $A \to A'$ *then* $, \vdash A = A' :: K$.

Proof: By structural induction on the derivation of $A \to A'$. Example cases:

R-DFUN: Given $\Pi x{:}A_1.\, A_2 \to \Pi x{:}A_1'.\, A_2'$ derived via rule R-DFUN from $A_1 \to A_1'$ and $A_2 \to A_2'$ as well as $, \vdash \Pi x{:}A_1.\, A_2 :: \Omega \Rightarrow\ , \vdash A_1 :: \Omega$ and $, , x{:}A_1 \vdash A_2 :: \Omega$ (C-DFUN). Applying the induction hypothesis gives us that $, \vdash A_1 = A_1' :: \Omega$ and $, , x{:}A_1 \vdash A_2 = A_2' :: \Omega$. Hence, by E-DFUN, $, \vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega$.

R-EXT: Given $x\rho_{A_1}\rho'\rho''! \to x\rho\rho'_{A_2}\rho''!$ derived via rule R-EXT from $A_1 \to A$ and $\mathcal{S}(A, x\rho, \rho') = A_2$ as well as $, \vdash x\rho_{A_1}\rho'\rho''! :: K$. By Theorem 7.6.15, $, \vdash A_1 :: \Omega$. Applying the induction hypothesis then gives us that $, \vdash A_1 = A :: \Omega$. The desired result then follows via the E-EXT2 rule.

R-BETA: Given $(\lambda\alpha{::}K.\, A_1)\, A_2 \to [A_2'/\alpha]A_1'$ derived via rule R-BETA from $A_1 \to A_1'$ and $A_2 \to A_2'$ as well as $, \vdash (\lambda\alpha{::}K.\, A_1)\, A_2 :: K' \Rightarrow\ , \vdash \lambda\alpha{::}K.\, A_1 :: K{\Rightarrow}K'$ and $, \vdash A_2 :: K$ (C-APP) $\Rightarrow\ , , \alpha{::}K \vdash A_1 :: K'$ (C-LAM). Applying the induction hypothesis then gives us that $, , \alpha{::}K \vdash A_1 = A_1' :: K'$ and $, \vdash A_2 = A_2' :: K$. Hence by E-LAM and E-APP, $, \vdash (\lambda\alpha{::}K.\, A_1)\, A_2 = (\lambda\alpha{::}K.\, A_1')\, A_2' :: K'$. By Theorem 7.6.10, $, , \alpha{::}K \vdash A_1' :: K'$ and $, \vdash A_2' :: K$. The desired result then follows via the E-BETA and E-TRAN rules.

R-ETA: Given $\lambda\alpha{::}K.\, A\, \alpha \to A'$ derived via rule R-ETA from $A \to A'$ and $\alpha \notin \mathrm{FCV}(A)$ as well as $, \vdash \lambda\alpha{::}K.\, A\, \alpha :: K{\Rightarrow}K' \Rightarrow\ , , \alpha{::}K \vdash A\, \alpha :: K'$ (C-LAM) $\Rightarrow\ , , \alpha{::}K \vdash A :: K{\Rightarrow}K'$ (C-APP). By Corollary 7.6.4, $, \vdash A :: K{\Rightarrow}K'$. The desired result then follows via the E-ETA rule.

R-ABBREV: Given $x\rho_A! \to A'$ derived via rule R-ABBREV from $A \to \texttt{<=}A'{::}K'\texttt{>}$ as well as $, \vdash x\rho_A! :: K$. By Theorem 7.6.15, $, \vdash A :: \Omega$. Applying the induction hypothesis then gives us that $, \vdash A = \texttt{<=}A'{::}K'\texttt{>} :: \Omega$. The desired result then follows via the E-ABBREV2 rule.

□

Corollary [Tagged] 7.6.17 (Subject reduction)
If $, \vdash A :: K$ *and* $A \to A'$ *then* $, \vdash A' :: K$.

Proof: By Theorem 7.6.16, $, \vdash A = A' :: K$. By Theorem 7.6.10, $, \vdash A' :: K$. □

7.7 Confluence

A *constructor context* C is a constructor with a single *hole*, written $[]$:

$$\begin{array}{lll} \mathit{Contexts}\ \ C & ::= & [] \mid \Pi x{:}C.\,A \mid \Pi x{:}A.\,C \mid \Sigma x{:}C.\,A \mid \Sigma x{:}A.\,C \mid \lambda\alpha{::}K.\,C \mid \\ & & C\,A \mid A\,C \mid {<}{=}C{::}K{>} \mid x\rho_C\rho'! \end{array}$$

The hole in C may be *filled* by a constructor A, written $C[A]$, by replacing the hole with A, incurring capture of free variables in A that are bound at the occurrence of the hole. For example, if $C = \lambda\alpha{::}\Omega.\,\Sigma x{:}{<}{=}\alpha{::}\Omega{>}.\,[]$, and $A = x_{<=\alpha::\Omega>}!$, then $C[A] = \lambda\alpha{::}\Omega.\,\Sigma x{:}{<}{=}\alpha{::}\Omega{>}.\,x_{<=\alpha::\Omega>}!$. The variables that are bound at the hole of a context are said to be *exposed (to capture)* by that context. I shall write ECV(C) for the exposed constructor variables and ETV(C) for the exposed term variables of a context.

Type checking is compositional in the sense that if a constructor is valid, then so are all its sub-constructors:

Lemma [Tagged] 7.7.1 (Decomposition)
Suppose that $,\ \vdash C[A] :: K$ *such that* $ETV(C) \cap dom(,\ ) = \emptyset$ *and* $ECV(C) \cap dom(,\ ) = \emptyset$.[1] *Then there exists* $,'$ *and* K' *such that:*

- $dom(,') = ECV(C) \cup ETV(C)$
- $,\ ;,' \vdash A :: K'$

Proof: By structural induction on C. Example cases:

DFUN I: Given $C = \Pi x{:}[].\,A_2$, $,\ \vdash \Pi x{:}A.\,A_2 :: \Omega$, ETV$(C) = \emptyset \cap \mathrm{dom}(,\ ) = \emptyset$, and ECV$(C) = \emptyset \cap \mathrm{dom}(,\ ) = \emptyset \Rightarrow$ ECV$(C) \cup$ ETV$(C) = \emptyset$ and $,\ \vdash A :: \Omega$ (C-DFUN). Thus, it suffices to let $,' = \bullet$ and $K' = \Omega$.

DFUN II: Given $C = \Pi x{:}A_1.\,[]$, $,\ \vdash \Pi x{:}A_1.\,A :: \Omega$, ETV$(C) = \{x\} \cap \mathrm{dom}(,\ ) = \emptyset$, and ECV$(C) = \emptyset \cap \mathrm{dom}(,\ ) = \emptyset$. $\Rightarrow x \notin \mathrm{dom}(,\ )$ and ECV$(C) \cup$ ETV$(C) = \{x\}$ and $,\ , x{:}A_1 \vdash A :: \Omega$ (C-DFUN). Thus, it suffices to let $,' = \bullet, x{:}A_1$ and $K' = \Omega$.

EXT: Given $C = x\rho_{[]}\rho'!$, $,\ \vdash x\rho_A\rho'! :: K$, ETV$(C) = \emptyset \cap \mathrm{dom}(,\ ) = \emptyset$, and ECV$(C) = \emptyset \cap \mathrm{dom}(,\ ) = \emptyset \Rightarrow$ ECV$(C) \cup$ ETV$(C) = \emptyset$. By Theorem 7.6.15, $,\ \vdash A :: \Omega$. Thus, it suffices to let $,' = \bullet$ and $K' = \Omega$.

□

[1]The conditions on the exposed variables can always be satisfied by alpha-renaming $C[A]$ appropriately.

Theorem [Tagged] 7.7.2 (Diamond property)
If $, \vdash A :: K$*,* $A \to A_1$*, and* $A \to A_2$ *then* $\exists A_3$ *such that* $A_1 \to A_3$ *and* $A_2 \to A_3$*.*

Proof: The proof is very similar to that of that for the erased system (Theorem 7.5.1), with the tagged rather than erased versions of the various lemmas and theorems being used. Lemma 7.7.1 is used to establish the validity of the various sub-constructors so that the induction hypothesis can be applied. Only one case requires different handling because of the presence of kind labels in lambdas:

- LAM;BETA (R) vs. ETA (WLOG $\alpha \neq \alpha'$; $\alpha \notin \mathrm{FCV}(A)$)

$$\begin{tikzcd}
& \lambda\alpha{::}K.\,(\lambda\alpha'{::}K'.\,A)\,\alpha \arrow[dl, "\mathrm{R}"'] \arrow[dr, "\mathrm{ETA}"] & & & \lambda\alpha'{::}K'.\,A \arrow[dl, dotted, "\mathrm{LAM}"'] \arrow[dr] & \\
\lambda\alpha{::}K.\,[\alpha/\alpha']A_1 \arrow[dr, dotted, "\Psi"'] & & A_2 \arrow[dl, dotted, "\Delta"] & \lambda\alpha'{::}K'.\,A_1 \arrow[dr, dotted, "\Psi\ (\mathrm{IH})"'] & & A_2 \arrow[dl, dotted, "\Delta\ (\mathrm{IH})"] \\
& A_3 & & & A_3 &
\end{tikzcd}$$

Here, $\lambda\alpha'{::}K'.\,A \to \lambda\alpha'{::}K'.\,A_1$ follows via the R-LAM rule from the given (not shown) $A \to A_1$. We are given that $, \vdash \lambda\alpha{::}K.\,(\lambda\alpha'{::}K'.\,A)\,\alpha :: K{\Rightarrow}K'' \Rightarrow$ $, , \alpha{::}K \vdash (\lambda\alpha'{::}K'.\,A)\,\alpha :: K''$ (C-LAM) $\Rightarrow$ $, , \alpha{::}K \vdash \lambda\alpha'{::}K'.\,A :: K{\Rightarrow}K''$ (C-APP and C-VAR) $\Rightarrow K = K'$ (C-LAM). Since $\alpha \notin \mathrm{FCV}(A_1)$ (by the reduct variables lemma), $\lambda\alpha'{::}K'.\,A_1 = \lambda\alpha'{::}K.\,A_1 = \lambda\alpha{::}K.\,[\alpha/\alpha']A_1$ (two constructors are considered the same if they are related by α-conversion).

□

Corollary [Tagged] 7.7.3 (Confluence)
If $, \vdash A :: K$*,* $A \to^* A_1$*, and* $A \to^* A_2$ *then* $\exists A_3$ *such that* $A_1 \to^* A_3$ *and* $A_2 \to^* A_3$*.*

Proof: Same proof as for the erased version (Theorem 7.5.2) except that the tagged rather than erased version of the diamond property is used and subject reduction (Corollary 7.6.17) is used to establish the validity precondition of the diamond property for the tagged system. □

7.8 Equality

Now that I have confluence, I can complete the proof that the rewriting relation implements equality:

Lemma [Tagged] 7.8.1 *If* , $\vdash x\rho_{A}\rho'! :: K$ *and* , $\vdash A = A' :: \Omega$ *then* , $\vdash x\rho_{A'}\rho'! :: K$.

Proof: By Lemma 7.6.5, , $\vdash x\rho \Rightarrow A$ and , $\vdash x\rho\rho' \Rightarrow A'$ where A' has either the form $<K>$ or the form $<=A''::K>$ for some A''. By P-MOVE, , $\vdash x\rho \Rightarrow A'$. Hence by C-EXT-O2 or C-EXT-T2, depending on the form of A'', the desired result follows. □

Theorem [Tagged] 7.8.2 *If* , $\vdash A = A' :: K$ *then* $\exists A_0, A_1, \ldots, A_n$, $n \geq 0$, *such that:*

1. $A = A_0$
2. $A' = A_n$
3. , $\vdash A_i = A_{i+1} :: K$ *for* $0 \leq i < n$
4. $A_i \rightleftharpoons A_{i+1}$ *for* $0 \leq i < n$

Proof: By structural induction on the typing derivation. Example cases:

SYM: Apply the induction hypothesis then use the E-SYM rule repeatedly to reverse all the equality judgments. Since bi-directional rewriting is symmetric, reversing the sequence of types obtained from the induction hypothesis then yields the desired sequence.

TRAN: Apply the induction hypothesis twice then place the two resulting sequences side by side so that A' is in the middle.

DFUN: Apply the induction hypothesis twice getting sequences $A'_0, \ldots, A'_m$ from , $\vdash A'_0 = A'_m :: \Omega$ and $A''_0, \ldots, A''_r$ from , $, x{:}A'_0 \vdash A''_0 = A''_r :: \Omega$. By Theorem 7.6.10, , $\vdash A'_i :: \Omega$ for $0 \leq i \leq m$ and , $, x{:}A'_0 \vdash A''_r :: \Omega$. Hence, by E-REFL, , $\vdash A'_0 = A'_0 :: \Omega$ and , $, x{:}A'_0 \vdash A''_r = A''_r :: \Omega$. By repeated use of the replacement with an equal type theorem (Theorem 7.6.9) then, , $, x{:}A'_i \vdash A''_r = A''_r :: \Omega$ for $0 \leq i < m$. The desired sequence is then $\Pi x{:}A'_0.\, A''_0, \ldots, \Pi x{:}A'_0.\, A''_r, \Pi x{:}A'_1.\, A''_r, \ldots, \Pi x{:}A'_m.\, A''_r$. E-DFUN is used connect them with equality judgments and R-DFUN and R-REFL to connect them with bi-directional rewriting.

BETA: Let $n = 1$. $A_0 \rightarrow A_1$ via R-BETA and R-REFL.

EXT2: Applying the induction hypothesis gives the sequence $A'_0, \ldots, A'_m$ from , $\vdash A_1 = A :: \Omega$. By Lemma 7.8.1, , $\vdash x\rho_{A'_i}\rho'\rho''! :: K$ for $0 \leq i \leq m$. The desired sequence is then $x\rho_{A'_0}\rho'\rho''!$, $\ldots$, $x\rho_{A'_m}\rho'\rho'!$, $x\rho\rho'_{A_2}\rho''!$. E-EXT2 is used connect them with equality judgments and R-EXT2 plus R-REFL to connect them with bi-directional rewriting.

ABBREV2: Applying the induction hypothesis gives the sequence $A'_0, \ldots, A'_m$ from $, \vdash A = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega$. By Lemma 7.8.1, $, \vdash x\rho_{A'_i}! :: K$ for $0 \leq i \leq m$. The desired sequence is then $x\rho_{A'_0}!, \ldots, x\rho_{A'_m}!, A'$. E-EXT2 plus E-ABBREV2 and E-REFL are used connect them with equality judgments and R-EXT plus R-ABBREV and R-REFL to connect them with bi-directional rewriting.

□

Theorem [Tagged] 7.8.3 (Church-Rosser)
If $n \geq 0$, $, \vdash A_i :: K$ for $0 \leq i \leq n$, and $A_i \rightleftharpoons A_{i+1}$ for $0 \leq i < n$ then $A_0 \downarrow^* A_n$.

Proof: Same basic proof as for the erased version (Theorem 7.5.3) except that the tagged rather than erased version of confluence is used and subject reduction (Corollary 7.6.17) is used to establish the validity precondition for tagged confluence. □

Corollary [Tagged] 7.8.4 (Partial Correctness III)
If $, \vdash A = A' :: K$ then $A \downarrow^* A'$.

Proof: Apply Theorem 7.8.2 followed by Theorem 7.8.3. □

Theorem [Tagged] 7.8.5 (Full correctness)
$, \vdash A_1 = A_2 :: K$ iff all of the following:

1. $, \vdash A_1 :: K$
2. $, \vdash A_2 :: K$
3. $A_1 \downarrow^* A_2$

Proof: The forward direction is handled by validity of equals (Theorem 7.6.10) and Corollary 7.8.4. The backwards direction is handled by repeated use of Theorem 7.6.16 and Theorem 7.6.10 followed by repeated use of E-TRAN, E-SYM, and E-REFL. □

An immediate corollary of this result is strengthening (removal of unreferenced declarations from an assignment) for equality:

Corollary [Tagged] 7.8.6 (Strengthening)
If $,_1, x{:}A; ,_2 \vdash A_1 = A_2 :: K$, $,_1; ,_2 \vdash A_1 :: K$, and $,_1; ,_2 \vdash A_2 :: K$ then $,_1; ,_2 \vdash A_1 = A_2 :: K$.

Proof: Follows immediately from Theorem 7.8.5. □

Combining the equality implementation result with shape preservation (Lemma 7.3.7) gives two important results about the properties of equality:

Lemma [Tagged] 7.8.7 (Equality of forms)
Suppose $, \vdash A_1 = A_2 :: K$ *and* A_2 *has neither the form* $A_2' \, A_2''$, $A_2' \notin \{\mathbf{ref}, \mathbf{rec}\}$, *the form* $x\rho_1{}_{A_1'}\rho_2!$, *nor the form* $\lambda\alpha{::}K.\, A_1'$. *Then*

1. *If* A_1 *has one of the forms* α, $<K>$, $\mathbf{rec}$, *or* $\mathbf{ref}$ *then* $A_1 = A_2$.
2. *If* A_1 *has the form* $\Pi x{:}A_1'.\, A_1''$ *then* A_2 *has the form* $\Pi x{:}A_2'.\, A_2''$.
3. *If* A_1 *has the form* $\Sigma x{:}A_1'.\, A_1''$ *then* A_2 *has the form* $\Sigma x{:}A_2'.\, A_2''$.
4. *If* A_1 *has the form* $<=A_1'{::}K>$ *then* A_2 *has the form* $<=A_2'{::}K>$.
5. *If* A_1 *has the form* $\mathbf{ref}\, A_1'$ *then* A_2 *has the form* $\mathbf{ref}\, A_2'$.
6. *If* A_1 *has the form* $\mathbf{rec}\, A_1'$ *then* A_2 *has the form* $\mathbf{rec}\, A_2'$.

Proof: By Theorem 7.8.5, $A_1 \downarrow^* A_2 \Rightarrow \exists A$ such that $A_1 \rightarrow^* A$ and $A_2 \rightarrow^* A$. By Lemma 7.3.7, A must have the same form as A_1. Moreover, by the same lemma, since all remaining forms possible for A_2 are preserved by rewriting, this means that A_2 can only have the same form as A (and hence, A_1). □

Lemma [Tagged] 7.8.8 (Component-wise equality)

1. *If* $, \vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega$, $x \notin dom(,)$, *then*
   $, \vdash A_1 = A_1' :: \Omega$ *and* $, , x{:}A_1 \vdash A_2 = A_2' :: \Omega$.
2. *If* $, \vdash \Sigma x{:}A_1.\, A_2 = \Sigma x{:}A_1'.\, A_2' :: \Omega$, $x \notin dom(,)$, *then*
   $, \vdash A_1 = A_1' :: \Omega$ *and* $, , x{:}A_1 \vdash A_2 = A_2' :: \Omega$.
3. *If* $, \vdash <K> = <K'> :: \Omega$ *then* $K = K'$.
4. *If* $, \vdash <=A{::}K> = <=A'{::}K'> :: \Omega$ *then*
   $K = K'$ *and* $, \vdash A = A' :: K$.
5. *If* $, \vdash \mathbf{ref}\, A = \mathbf{ref}\, A' :: \Omega$ *then* $, \vdash A = A' :: \Omega$.
6. *If* $, \vdash \mathbf{rec}\, A = \mathbf{rec}\, A' :: \Omega$ *then* $, \vdash A = A' :: \Omega \Rightarrow \Omega$.

Proof: The proofs of each part are similar; we give here only the proof for part one: Given , $\vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega$. By Theorem 7.8.5, $\Pi x{:}A_1.\, A_2 \downarrow^* \Pi x{:}A_1'.\, A_2'$, , $\vdash \Pi x{:}A_1.\, A_2 :: \Omega$, and , $\vdash \Pi x{:}A_1'.\, A_2' :: \Omega \Rightarrow$, , $x{:}A_1 \vdash A_2 :: \Omega$ and , , $x{:}A_1' \vdash A_2' :: \Omega$, (C-DFUN) and $\exists A''$ such that $\Pi x{:}A_1.\, A_2 \rightarrow^* A''$ and $\Pi x{:}A_1'.\, A_2' \rightarrow^* A'' \Rightarrow$, $\vdash A_1 :: \Omega$ and , $\vdash A_1' :: \Omega$ (Theorem 6.8.3 and DECL-T). Hence, by shape preservation (Lemma 7.3.7), $\exists A_1'', A_2''$ such that $A'' = \Pi x{:}A_1''.\, A_2''$, $A_1 \rightarrow^* A_1''$, $A_1' \rightarrow^* A_1''$, $A_2 \rightarrow^* A_2''$, and $A_2' \rightarrow^* A_2''$. By Theorem 7.8.5 and Corollary 7.6.17 then, , $\vdash A_1 = A_1' :: \Omega$, , , $x{:}A_1 \vdash A_2 = A_2'' :: \Omega$, and , , $x{:}A_1' \vdash A_2' = A_2'' :: \Omega \Rightarrow$, , $x{:}A_1 \vdash A_2' = A_2'' :: \Omega$ (Theorem 7.6.9) $\Rightarrow$, , $x{:}A_1 \vdash A_2 = A_2' :: \Omega$ (E-TRAN). $\square$

Chapter 8

Types: Canonicalization

In this chapter, I show how to compute normal forms for valid tagged constructors and how to decide all tagged judgments. These results, once transfered back to the original untagged system (see the next chapter), will be crucial in proving the (semi)-decidability of subtyping (and hence type checking in general). All constructors, assignments, etc., in this chapter are tagged unless stated otherwise.

8.1 Normal Forms

Definition [Tagged] 8.1.1 (Normal form)
A is in normal form for rewriting relation R (often written R-normal form or just normal form if the rewriting relation is the full one) iff $\nexists A'.\ A \rightarrow^1_R A'$.

If $A \rightarrow^*_R A'$ and A' is in R-normal form, then A' is said to be a R-normal form for A and A is said to be R-normalizing. If all reduction sequences starting from A using $\rightarrow^1_R$ eventually lead to a R-normal form, then A is said to be *strongly R-normalizing*:

Definition [Tagged] 8.1.2 (Strong normalization)
A is strongly normalizing under R-reduction iff there are no infinite reduction sequences starting from A using $\rightarrow^1_R$ (i.e., $A \rightarrow^1_R A_1 \rightarrow^1_R A_2 \rightarrow^1_R A_3 \ldots$).

(I am using $\rightarrow^1_R$ instead of $\rightarrow_R$ here to rule out steps, such as that generated by R-REFL, which leave the constructor unchanged.)

One advantage of normal forms is that they allow checking for a common reduct to be reduced to an easier check for identity (modulo α-conversion):

Lemma [Tagged] 8.1.3 *If A_1 and A_2 are both in normal form then $A_1 \downarrow^* A_2$ iff $A_1 = A_2$.*

Proof: By the definition of a common reduct, $A_1 \downarrow^* A_2$ iff $\exists A.\ A_1 \to^* A$ and $A_2 \to^* A$. Since A_i is in normal form, $A_i \to^* A_i'$ implies that $A_i = A_i'$. Hence, $A_1 \downarrow^* A_2$ iff $\exists A.\ A_1 = A$ and $A_2 = A \Rightarrow A_1 \downarrow^* A_2$ iff $A_1 = A_2$. □

This fact when combined with the implementing equality result from the previous chapter means that I can check that $, \vdash A_1 = A_1 :: K$ by checking that $\forall i.\ , \vdash A_i :: K$ and that A_1 and A_2 have the same normal form.

Another similar advantage of normal forms is that they make searching for constructors equal to a given constructor but with a certain form easy. For example, suppose we wanted to know if A under , is equal to a constructor of the form $<=A_1::K>$ for some K and A_1, given $, \vdash A :: \Omega$ and A' a normal form for A. We can reason as follows:

Assume such a constructor exists. Then, by the implementing equality result, $A' \downarrow^* <=A_1::K> \Rightarrow <=A_1::K> \to^* A'$ since A' is in normal form $\Rightarrow A'$ must have the form $<=A_1'::K>$ for some A_1' since rewriting preserves this shape (see Lemma 7.3.7). Thus, we have that A' itself must have the form in question iff such an equal constructor exists. Note that this new condition can be checked by a simple inspection and that the inspection provides an example of such an equal constructor.

Because my rewriting relation allows only $\beta\eta$- and γ-reductions, a constructor is in normal form iff it is in both $\beta\eta$- and γ-normal form:

Lemma [Tagged] 8.1.4 *If $A \to^1 A'$ then $\exists A''$ such that either $A \to^1_{\beta\eta} A''$ or $A \to^1_{\gamma} A''$.*

Proof: By structural induction on the derivation of $A \to^1 A'$. Example cases:

APP: Given $A_1\,A_2 \to^1 A_1'\,A_2'$ derived via rule R-APP from $A_1 \to A_1'$ and $A_2 \to A_2'$. Since $A_1\,A_2 \neq A_1'\,A_2'$, we must have $A_i \to^1 A_i'$ for some i in $\{1, 2\}$. Applying the induction hypothesis gives us that $\exists A_i''$ such that either $A_i \to^1_{\beta\eta} A_i''$ or $A_i \to^1_{\gamma} A_i''$. The desired result follows by R-APP and R-REFL. (A'' is either $A_1\,A_2''$ or $A_1''\,A_2$.)

EXT I: Given $x\rho_{A_1}\rho'\rho''! \to^1 x\rho\rho'_{A_2}\rho''!$, $\rho' \neq \epsilon$, derived via rule R-EXT from $A_1 \to A_1'$ and $\mathcal{S}(A_1', x\rho, \rho') = A_2$. If $A_1 = A_1'$ then $x\rho_{A_1}\rho'\rho''! \to^1_{\gamma} x\rho\rho'_{A_2}\rho''!$ via R-EXT and R-REFL and we are done. Otherwise, $A_1 \to^1 A_1'$. Applying the induction hypothesis gives us that $\exists A_1''$ such that either $A_1 \to^1_{\beta\eta} A_1''$ or $A_1 \to^1_{\gamma} A_1''$. By R-EXT then, either $x\rho_{A_1}\rho'\rho''! \to^1_{\beta\eta} x\rho_{A_1''}\rho'\rho''!$ or $x\rho_{A_1}\rho'\rho''! \to^1_{\gamma} x\rho_{A_1''}\rho'\rho''!$.

ABBREV: Given $x\rho_{A_1}! \to^1 A_1'$ derived via rule R-ABBREV from $A_1 \to <=A_1'::K'>$. If $A_1 = <=A_1'::K'>$ then $x\rho_{A_1}! \to^1_{\gamma} A_1'$ via R-ABBREV and R-REFL so we are done. Otherwise, $A_1 \to^1 <=A_1'::K'>$. Applying the induction hypothesis gives us that $\exists A_1''$ such that either $A_1 \to^1_{\beta\eta} A_1''$ or $A_1 \to^1_{\gamma} A_1''$. Hence, by R-EXT, either $x\rho_{A_1}! \to^1_{\gamma} x\rho_{A_1''}!$ or $x\rho_{A_1}! \to^1_{\beta\eta} x\rho_{A_1''}!$.

$\square$

Corollary [Tagged] 8.1.5 *If A is in both $\beta\eta$-normal form and γ-normal form then A is in normal form.*

Proof: Assume not. Then $\exists A'.\ A \rightarrow^1 A'$. Hence, by Lemma 8.1.4, $\exists A''$ such that either $A \rightarrow^1_{\beta\eta} A''$ or $A \rightarrow^1_{\gamma} A''$. But this is impossible because A is in both $\beta\eta$-normal form and γ-normal form. $\square$

8.2 The Method

As with F_ω, invalid constructors in my system may not be normalizing. For example, if $W = \lambda\alpha{::}\Omega.\,(\alpha\,\alpha)\,(\alpha\,\alpha)$ then $W\,W$ is not normalizing because we can apply the step $W\,W \rightarrow_{\beta\eta} (W\,W)\,(W\,W)$ to sub-constructors of it forever. Thus, in general normalization of a constructor in my system will require that the constructor be valid.

In F_ω, which has only β- and η-reductions, valid constructors are strongly normalizing. This result can be proved by encoding F_ω's kind and constructor levels into the type and term levels of the simply-typed lambda calculus ($\lambda^{\rightarrow}$) and then using the fact that $\lambda^{\rightarrow}$ is known to be strongly normalizing for well-typed terms. The same method (see Section 8.3) can be used to prove that valid constructors in the tagged system are strongly $\beta\eta$-normalizing. Handling normalization for the full rewriting relation is more difficult.

It is not possible in the tagged system to first check that a constructor is valid and then normalize it by reducing it until it can no longer be reduced any further. The reason is that validity checking requires checking equality on sub-constructors which in turn requires normalizing sub-constructors. Accordingly, it is necessary to interleave the tasks of checking a constructor's validity and computing its normal form in a single algorithm that returns a normal form only when the constructor is valid. Because constructors can have at most one kind (Theorem 7.6.11), the algorithm can also return the constructor's kind when it is valid.

Such an algorithm can be defined inductively over $(, , A)$ pairs (, is needed to check variable references). For example, to handle the case where $A = \Sigma\alpha{:}A_1.\,A_2$, $\alpha \notin \mathrm{dom}(,)$, we can first call ourselves recursively on $(, , A_1)$ and $((, , \alpha{:}A_1), A_2)$. If A_1 or A_2 are found to be invalid or to have kinds other than Ω, $\Sigma\alpha{:}A_1.\,A_2$ must be invalid as well. Otherwise, the desired normal form is $\Sigma\alpha{:}A_1'.\,A_2'$ ($\Sigma\alpha{:}A_1'.\,A_2'$ is in normal form iff A_1' and A_2' are) and the kind of A is Ω.

There are two problematic cases. The first one is constructor extraction where $A = x\rho_{A_1}\rho'!$. We need to check that $, \vdash x\rho \Rightarrow A_1$ and search for a kind K and constructor A_2 in normal form such that either $, \vdash x\rho\rho' \Rightarrow {<}K{>}$ or $, \vdash x\rho\rho' \Rightarrow {<}{=}A_2{::}K{>}$. Only if both the check and the search succeed is A valid; in that case A will have kind K and normal form $x\rho\rho'_{<K>}!$ (if $\Gamma \vdash x\rho\rho' \Rightarrow <K>$) or A_2 (otherwise).

By combining the properties of normal forms with parallelism II (Theorem 7.3.13), it can be shown that if A_0 is a normal form for $\Gamma(x)$ then $\Gamma \vdash x\rho \Rightarrow A'$ iff $\mathcal{S}(A_0, x, \rho)$ exists and $\Gamma \vdash A' = \mathcal{S}(A_0, x, \rho) :: \Omega$. This result allows handling the above questions, as well as place lookup judgments in general, via questions about equality that can be handled using the methods of the previous section.

The second problematic case is constructor application where $A = A_1\, A_2$. We can first call ourselves recursively on (Γ, A_1) and (Γ, A_2). A has kind K (and hence is valid) iff A_1 has kind $K' \Rightarrow K$ and A_2 has kind K' for some K'. This condition is easily checked; finding a normal form for A when it is valid is more difficult. The recursive calls will have given us a normal form for A_1 (call it A_1') and a normal form for A_2 (call it A_2'). Hence, $A \rightarrow^* A_1'\, A_2'$ by R-APP.

Unfortunately, while $A_1'\, A_2'$ is in γ-normal form, it is not necessarily in $\beta\eta$-normal form (e.g., consider $(\lambda\alpha{::}\Omega.\, \alpha)\,{<}\Omega{>}$). At this point, we do know that $A_1'\, A_2'$ is a valid constructor so we could repeatedly $\beta\eta$-reduce it to a $\beta\eta$-normal form. (This process must terminate since valid constructors are strongly $\beta\eta$-normalizing.) It is not immediately clear, however, if this procedure works in general because the $\beta\eta$-reductions might enable new γ-reductions, resulting in a constructor not in γ-normal form.

For example, β-reducing the following constructor enables an γ-reduction:

$$(\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\, x_{\alpha\,<=<\Omega>::\Omega>}!)\; \lambda\alpha'{::}\Omega.\, \alpha'$$

This and other such examples are not valid constructors however (the constructor extraction is invalid because $\alpha\,{<}{=}{<}\Omega{>}{::}\Omega{>}$ cannot be shown equal to any reified constructor type) which suggests that there may be some special property of these constructors that rules out the enabling of new γ-reductions by $\beta\eta$-reductions.

This intuition is in fact correct and can be seen by looking again at discussion of the first problematic case. Notice that we introduce constructor extractions of only the form $x\rho_{<K>}!$. I shall call a constructor that contains constructor extractions of only this form, a constructor in *γ-final form.* Such constructors must trivially be in γ-normal form. More interestingly, this property is stable under $\beta\eta$-reductions because the constructor extractions contain no free constructor variables, which could be substituted for by the $\beta\eta$-reductions.

Thus, if we knew that A_1' and A_2' were in γ-final form, we could safely just $\beta\eta$-normalize their application and return the result as a normal form for A. In order to get A_1' and A_2' in γ-final form, we need to show that our algorithm returns not just normal forms but normal forms in γ-final form (*canonical forms*). This result can be established by using the obvious stronger induction hypothesis when proving the algorithm correct.

8.3 $\beta\eta$-Normalization

In this section I prove that valid constructors are strongly $\beta\eta$-normalizing and give an algorithm, using this result, that $\beta\eta$-normalizes valid constructors. I prove the result by encoding kinds and constructors into $\lambda^{\rightarrow}$'s type and term levels respectively and then taking advantage of $\lambda^{\rightarrow}$'s strong normalization property.

The syntax for the $\lambda^{\rightarrow}$ system is quite simple:

Definition [$\lambda^{\rightarrow}$] 8.3.1 (Syntax)

Types	A	$::=$	$\Omega \mid A {\rightarrow} A'$
Terms	M	$::=$	$x \mid \Psi_A \mid \lambda x{:}A.\, M \mid M\, M'$
Assignments	,	$::=$	$\bullet \mid$, , $x{:}A$

Here, the metavariable x ranges over *term variables* and Ψ_A denotes an uninterpreted constant of type A. (I have included the constants solely to make the encoding simpler; a more complicated proof can be constructed without them.)

The usual conventions apply and we have the usual operators:

Definition [$\lambda^{\rightarrow}$] 8.3.2 (Free term variables)

$$\begin{array}{rcl}
FTV(x) & = & \{x\} \\
FTV(\Psi_A) & = & \emptyset \\
FTV(\lambda x{:}A.\, M) & = & FTV(M) \Leftrightarrow \{x\} \\
FTV(M_1\, M_2) & = & FTV(M_1) \cup FTV(M_2)
\end{array}$$

Definition [$\lambda^{\rightarrow}$] 8.3.3 (Term substitution)

$$\begin{array}{rcll}
[M/x]x & = & M & \\
{[M/x]x'} & = & x' & (x \neq x') \\
\\
{[M/x]\Psi_A} & = & \Psi_A & \\
{[M/x]\lambda x'{:}A.\, M'} & = & \lambda x'{:}A.\, [M/x]M' & (x' \neq x,\ x' \notin FTV(M)) \\
{[M/x](M_1\, M_2)} & = & [M/x]M_1\, [M/x]M_2 &
\end{array}$$

Definition [$\lambda^{\rightarrow}$] 8.3.4 (Assignment regarded as a partial function)

$$\begin{array}{rcll}
dom(\bullet) & = & \emptyset & \\
dom(,\ , x{:}A) & = & dom(,\ ) \cup \{x\} & \\
\\
(,\ _1; x{:}A; ,\ _2)(x) & = & A & (x \notin dom(,\ _1))
\end{array}$$

The system has only two judgments, which have simple rules:

Definition [$\lambda^{\rightarrow}$] 8.3.5 (Judgments)

$$\begin{array}{ll} \vdash ,\ \ \mathit{valid} & \textit{valid assignment} \\ ,\ \vdash M : A & \textit{well-typed term} \end{array}$$

Definition [$\lambda^{\rightarrow}$] 8.3.6 (Assignment Formation Rules)

$$\vdash \bullet\ \mathit{valid} \qquad \text{(L-EMPTY)}$$

$$\frac{\vdash ,\ \ \mathit{valid} \qquad x \notin \mathit{dom}(,\ )}{\vdash ,\ , x{:}A\ \mathit{valid}} \qquad \text{(L-EXTEND)}$$

Definition [$\lambda^{\rightarrow}$] 8.3.7 (Term Formation Rules)

$$\frac{\vdash ,\ \ \mathit{valid} \qquad x{:}A \in ,}{,\ \vdash x : A} \qquad \text{(L-VAR)}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \Psi_{A} : A} \qquad \text{(L-CON)}$$

$$\frac{,\ , x{:}A \vdash M : A'}{,\ \vdash \lambda x{::}A.\, M : A{\rightarrow}A'} \qquad \text{(L-LAM)}$$

$$\frac{,\ \vdash M_1 : A_2{\rightarrow}A \qquad ,\ \vdash M_2 : A_2}{,\ \vdash M_1\, M_2 : A} \qquad \text{(L-APP)}$$

The rewriting relation for $\lambda^{\rightarrow}$ is defined on terms rather than types and permits both β- and η-reductions. Its rules are as follows:

Definition [$\lambda^{\rightarrow}$] 8.3.8 (Rewrite relation)

$$\frac{M \rightarrow M'}{\lambda x{:}A.\, M \rightarrow \lambda x{:}A.\, M'} \qquad \text{(LR-LAM)}$$

$$\frac{M_1 \rightarrow M_1'}{M_1\, M_2 \rightarrow M_1'\, M_2} \qquad \text{(LR-LEFT)}$$

$$\frac{M_2 \rightarrow M_2'}{M_1\, M_2 \rightarrow M_1\, M_2'} \qquad \text{(LR-RIGHT)}$$

$$(\lambda x{:}A.\, M_1)\, M_2 \rightarrow [M_2/x]M_1 \qquad \text{(LR-BETA)}$$

$$\frac{x \notin FTV(M)}{\lambda x{:}A.\, M\, x \rightarrow M} \qquad \text{(LR-ETA)}$$

One of the classic results for $\lambda^{\rightarrow}$ (see, for example, Barendregt [2]) is that all well-typed terms are strongly normalizing:

Theorem [$\lambda^{\rightarrow}$] 8.3.9 (Strong normalization)
If , $\vdash M : A$ then there are no infinite reduction sequences starting from A using $\rightarrow$ (i.e., $A \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \ldots$).

The encoding I will be using is follows. It encodes kinds as $\lambda^{\rightarrow}$-types, constructor variables as $\lambda^{\rightarrow}$-term variables, assignments as $\lambda^{\rightarrow}$-assignments, and constructors as $\lambda^{\rightarrow}$-terms. The encoding of constructors takes the constructor's assignment as an extra argument so that it can determine the kind of constructor extractions; this information is necessary to ensure that the encoding maps valid constructors to well-typed terms.

Definition [Tagged] 8.3.10 (The $\lambda^{\rightarrow}$-encoding)

$$\begin{array}{rcll}
\Omega^{\star} & = & \Omega & \\
(K_1{\Rightarrow}K_2)^{\star} & = & K_1^{\star}{\rightarrow}K_2^{\star} & \\
 & & & \\
\bullet^{\star} & = & \bullet & \\
(,\, ,\alpha{::}K)^{\star} & = & ,\, ^{\star}, \alpha^{\star}{:}K^{\star} & \\
(,\, ,x{:}A)^{\star} & = & ,\, ^{\star} & \\
 & & & \\
\alpha^{\star,} & = & \alpha^{\star} & \\
(\lambda\alpha{::}K.\, A)^{\star,} & = & \lambda\alpha^{\star}{:}K^{\star}.\, A^{\star,\ ,\alpha::K} & (\alpha \notin dom(,\,)) \\
(A_1\, A_2)^{\star,} & = & A_1^{\star,}\ A_2^{\star,} & \\
 & & & \\
(\Pi x{:}A_1.\, A_2)^{\star,} & = & \Psi_{\Omega\rightarrow\Omega\rightarrow\Omega}\, A_1^{\star,}\ A_2^{\star,\ ,x:A_1} & (x \notin dom(,\,)) \\
(\Sigma x{:}A_1.\, A_2)^{\star,} & = & \Psi_{\Omega\rightarrow\Omega\rightarrow\Omega}\, A_1^{\star,}\ A_2^{\star,\ ,x:A_1} & (x \notin dom(,\,)) \\
{<}K{>}^{\star,} & = & \Psi_{\Omega} & \\
{<}{=}A{::}K{>}^{\star,} & = & \Psi_{K^{\star}\rightarrow\Omega}\, A^{\star,} & \\
x\rho_A\rho'!^{\star,} & = & \Psi_{\Omega\rightarrow K^{\star}}\, A^{\star,} & \textit{where } ,\, \vdash x\rho_A\rho'! :: K \\
\mathbf{rec}^{\star,} & = & \Psi_{(\Omega\rightarrow\Omega)\rightarrow\Omega} & \\
\mathbf{ref}^{\star,} & = & \Psi_{\Omega\rightarrow\Omega} &
\end{array}$$

(The mapping from constructor variables to $\lambda^{\rightarrow}$ term variables is not shown; any bijective mapping is sufficient.)

The basic idea of the encoding is to map variables, functions, and applications to $\lambda^{\rightarrow}$-terms of the corresponding sort; all other sorts of constructors are mapped to constants or applications of constants to the encodings of the constructor's component constructors. This mapping allows any $\beta\eta$-reduction between two constructors to be mirrored on the corresponding $\lambda^{\rightarrow}$-terms. Note that the encoding of constructors is well defined when restricted to valid constructors:

Lemma [Tagged] 8.3.11 (Existence)
If $, \vdash A :: K$ *then* $A^{\star,}$ *exists and is uniquely defined (modulo* α*-conversion).*

Proof: The only case which could cause $A^{\star,}$ to be undefined or multiply defined is if a sub-constructor of A of the form $x\rho_{A'}\rho'!$ is either not well-formed or has multiple kinds. However, this is not possible: by decomposition (Lemma 7.7.1), all such sub-constructors must be well-formed and by Lemma 7.6.11, they each have only one kind. □

The encoding possesses two key properties. The first is that it maps valid assignments to $\lambda^{\rightarrow}$ valid assignments and constructors to $\lambda^{\rightarrow}$ well-typed terms:

Lemma [Tagged] 8.3.12

1. *If* $\vdash ,$ *valid then* $\vdash ,^{\star}$ *valid.*

2. *If* $, \vdash A :: K$ *then* $,^{\star} \vdash A^{\star,} : K^{\star}$.

Proof: Proved sequentially by structural induction on , and A respectively. Example cases:

DECL-C: Given $\vdash , , \alpha{::}K$ valid derived via rule DECL-C from $\vdash ,$ valid and $\alpha \notin \mathrm{dom}(,)$. Applying the induction hypothesis gives us that $\vdash ,^{\star}$ valid. Inspection of the encoding reveals that $\alpha' \in \mathrm{dom}(,)$ iff $\alpha'^{\star} \in \mathrm{dom}(,^{\star}) \Rightarrow \alpha^{\star} \notin \mathrm{dom}(,^{\star})$. Hence, by L-EXTEND, $\vdash ,^{\star}, \alpha^{\star}{:}K^{\star}$ valid
$\Rightarrow \vdash (, , \alpha{::}K)^{\star}$ valid.

C-DFUN: Given $, \vdash \Pi x{:}A_1.\, A_2 :: \Omega$, $x \notin \mathrm{dom}(,)$, derived via rule C-DFUN from $, , x{:}A_1 \vdash A_2 :: \Omega$ $\Rightarrow , \vdash A_1 :: \Omega$ and $\vdash ,$ valid (Theorem 6.8.3). By part one, $\vdash ,^{\star}$ valid. Applying the induction hypothesis gives us that $,^{\star} \vdash A_1^{\star,} : \Omega$ and $,^{\star} \vdash A_2^{\star, ,x:A_1} : \Omega$. Hence, by rules L-CON and L-APP, $,^{\star} \vdash \Psi_{\Omega\rightarrow\Omega\rightarrow\Omega}\, A_1^{\star,}\; A_2^{\star, ,x:A_1} : \Omega \Rightarrow$
$,^{\star} \vdash (\Pi x{:}A_1.\, A_2)^{\star,} : \Omega^{\star}$.

C-LAM: Given $, \vdash \lambda\alpha{::}K.\,A :: K{\Rightarrow}K'$, $\alpha \notin \mathrm{dom}(,)$, derived via rule C-LAM from $, ,\alpha{::}K \vdash A :: K'$. Applying the induction hypothesis gives us that
$,^{\star}, \alpha^{\star}{:}K^{\star} \vdash A^{\star, ,\alpha::K} : K'^{\star} \Rightarrow ,^{\star} \vdash \lambda\alpha^{\star}{:}K^{\star}.\,A^{\star, ,\alpha::K} : K^{\star}{\to}K'^{\star}$ (L-LAM)
$\Rightarrow ,^{\star} \vdash (\lambda\alpha{::}K.\,A)^{\star,} : (K{\Rightarrow}K')^{\star}$.

C-EXT-T2: Given $, \vdash x\rho_{A}\rho'! :: K \Rightarrow \vdash ,$ valid (Theorem 6.8.3) $\Rightarrow \vdash ,^{\star}$ valid (part one). By Theorem 7.6.15, $, \vdash A :: \Omega$. Applying the induction hypothesis gives us that
$,^{\star} \vdash A^{\star,} : \Omega$. Hence, by rules L-CON and L-APP, $,^{\star} \vdash \Psi_{\Omega\to K^{\star}}\,A^{\star,} : K^{\star} \Rightarrow$
$,^{\star} \vdash x\rho_{A}\rho'!^{\star,} : K^{\star}$.

□

The second key property of the encoding is that two constructors related by a $\beta\eta$-reduction are still related by a reduction after they are encoded as $\lambda^{\rightarrow}$-terms; moreover, irreflexive rewriting steps are mirrored by one or more $\lambda^{\rightarrow}$ rewriting steps. (The later part is needed to ensure that mapping an infinite reduction sequence of irreflexive steps into $\lambda^{\rightarrow}$ generates an infinite reduction sequence.)

Lemma [Tagged] 8.3.13 *If $A^{\star,}$ exists then $FTV(A^{\star,}) = FCV(A)^{\star}$.*

Lemma [Tagged] 8.3.14 (Encoding properties)

1. *If $,_1; ,_3 \vdash A :: K$, $\vdash ,_1; ,_2$ valid, and $dom(,_2) \cap dom(,_3) = \emptyset$ then* $A^{\star, _1;, _2;, _3} = A^{\star, _1;, _3}$.

2. *If $,_1 \vdash A_1 :: K$ and $,_1, \alpha{::}K; ,_2 \vdash A_2 :: K'$ then*

$$[A_1^{\star, _1}/\alpha^{\star}](A_2^{\star, _1,\alpha::K;, _2}) = ([A_1/\alpha]A_2)^{\star, _1;[A_1/\alpha], _2}$$

Proof: Proved sequentially using structural induction on A and A_2. Part one requires the use of weakening (Theorem 6.8.10). Part two requires the use of part one, Theorem 7.6.2, and Lemma 6.8.3. □

Lemma [Tagged] 8.3.15 (Mirroring) *Suppose $, \vdash A_1 :: K$. Then:*

1. *If $A_1 \to^{1}_{\beta\eta} A_2$ then $A_1^{\star,} \to^{+} A_2^{\star,}$.*

2. *If $A_1 \to_{\beta\eta} A_2$ then $A_1^{\star,} \to^{*} A_2^{\star,}$.*

Proof: Note that the second part follows immediately from the first part plus the fact that $\rightarrow^*$ is reflexive; it will accordingly be used recursively in the proof of the first part.

The proof of part one proceeds by structural induction on the derivation of $A_1 \rightarrow^1_{\beta\eta} A_2$. By Lemma 8.3.12 and subject reduction (Corollary 7.6.17), $A_1^{\star ,}$ and $A_2^{\star ,}$ exist. Example cases:

R-DFUN: Given , $\vdash \Pi x{:}A_1.\, A_2 :: \Omega$ and $\Pi x{:}A_1.\, A_2 \rightarrow^1_{\beta\eta} \Pi x{:}A_1'.\, A_2'$, $x \notin$ dom(,), derived via rule R-DFUN from $A_1 \rightarrow_{\beta\eta} A_1'$ and $A_2 \rightarrow_{\beta\eta} A_2' \Rightarrow$, $\vdash A_1 :: \Omega$ and , , $x{:}A_1 \vdash A_2 :: \Omega$ (Theorem 6.8.3), and $A_1 \rightarrow^1_{\beta\eta} A_1'$ or $A_2 \rightarrow^1_{\beta\eta} A_2'$.

Applying the induction hypothesis gives us that $A_1^{\star ,} \rightarrow^+ A_1'^{\star ,}$ and $A_2^{\star , ,x:A_1} \rightarrow^* A_2'^{\star , ,x:A_1'}$ or that $A_1^{\star ,} \rightarrow^* A_1'^{\star ,}$ and $A_2^{\star , ,x:A_1} \rightarrow^+ A_2'^{\star , ,x:A_1'}$

$\Rightarrow A_1^{\star ,}\ A_2^{\star , ,x:A_1} \rightarrow^+ A_1'^{\star ,}\ A_2^{\star , ,x:A_1} \rightarrow^* A_1'^{\star ,}\ A_2'^{\star , ,x:A_1'}$ or $A_1^{\star ,}\ A_2^{\star , ,x:A_1} \rightarrow^*$ $A_1'^{\star ,}\ A_2^{\star , ,x:A_1} \rightarrow^+ A_1'^{\star ,}\ A_2'^{\star , ,x:A_1'}$ (repeated use of LR-LEFT followed by LR-RIGHT)

$\Rightarrow A_1^{\star ,}\ A_2^{\star , ,x:A_1} \rightarrow^+ A_1'^{\star ,}\ A_2'^{\star , ,x:A_1'}$

$\Rightarrow \Psi_{\Omega\rightarrow\Omega\rightarrow\Omega}\, A_1^{\star ,}\ A_2^{\star , ,x:A_1} \rightarrow^+ \Psi_{\Omega\rightarrow\Omega\rightarrow\Omega}\, A_1'^{\star ,}\ A_2'^{\star , ,x:A_1'}$ (L-RIGHT)

$\Rightarrow (\Pi x{:}A_1.\, A_2)^{\star ,} \rightarrow^+ (\Pi x{:}A_1'.\, A_2')^{\star ,}$

R-BETA: Given , $\vdash (\lambda\alpha{::}K.\, A_1)\, A_2 :: K'$ and $(\lambda\alpha{::}K.\, A_1)\, A_2 \rightarrow^1_{\beta\eta} [A_2'/\alpha]A_1'$, $\alpha \notin$ dom(,), derived via rule R-BETA from $A_1 \rightarrow_{\beta\eta} A_1'$ and $A_2 \rightarrow_{\beta\eta} A_2' \Rightarrow$, $\vdash A_2 :: K$ and , , $\alpha{::}K \vdash A_1 :: K'$ (C-APP and C-LAM).

Applying the induction hypothesis gives us that $A_1^{\star , ,\alpha::K} \rightarrow^* A_1'^{\star , ,\alpha::K}$ and $A_2^{\star ,} \rightarrow^* A_2'^{\star ,}$. Hence, $((\lambda\alpha{::}K.\, A_1)\, A_2)^{\star ,} = (\lambda\alpha^\star{:}K^\star.\, A_1^{\star , ,\alpha::K})\, A_2^{\star ,}$ $\rightarrow^* (\lambda\alpha^\star{:}K^\star.\, A_1'^{\star , ,\alpha::K})\, A_2'^{\star ,} \rightarrow [A_2'^{\star ,}/\alpha](A_1'^{\star , ,\alpha::K}) = ([A_2'/\alpha]A_1')^{\star ,}$ (the last by Lemma 8.3.14).

R-ETA: Given , $\vdash \lambda\alpha{::}K.\, A\,\alpha :: K{\Rightarrow}K'$ and $\lambda\alpha{::}K.\, A\,\alpha \rightarrow^1_{\beta\eta} A'$, $\alpha \notin$ dom(,), derived via rule R-ETA from $A \rightarrow_{\beta\eta} A'$ and $\alpha \notin \mathrm{FCV}(A)$

$\Rightarrow$, , $\alpha{::}K \vdash A :: K{\Rightarrow}K'$ (C-LAM and C-APP) $\Rightarrow$, $\vdash A :: K{\Rightarrow}K'$ (Corollary 7.6.4). Applying the induction hypothesis then gives us that $A^{\star ,} \rightarrow^* A'^{\star ,}$. By Lemma 8.3.13, $\alpha^\star \notin \mathrm{FTV}(A^{\star ,})$. Hence, $(\lambda\alpha{::}K.\, A\,\alpha)^{\star ,} = \lambda\alpha^\star{:}K^\star.\, A^{\star , ,\alpha::K}\,\alpha = \lambda\alpha^\star{:}K^\star.\, A^{\star ,}\,\alpha \rightarrow$ $A^{\star ,} \rightarrow^* A'^{\star ,}$ (the second equality is by Lemma 8.3.14).

□

Using these two properties of the encoding, a proof of strong $\beta\eta$-normalization for valid constructors is easily constructed:

Theorem [Tagged] 8.3.16 (Strong $\beta\eta$-normalization)
If , $\vdash A :: K$ *then* A *is strongly normalizing under* $\beta\eta$*-reduction.*

Proof: By Lemma 8.3.12, , $^{\star} \vdash A^{\star}$' $: K^{\star}$. Suppose A is not strongly normalizing under $\beta\eta$-reduction. Then there exists an infinite $\beta\eta$-reduction sequence starting from A, say $A \rightarrow^1_{\beta\eta} A_1 \rightarrow^1_{\beta\eta} A_2 \rightarrow^1_{\beta\eta} A_3 \ldots$. Hence by repeated use of Lemma 8.3.15 and subject reduction (Corollary 7.6.17), $A^{\star}$' $\rightarrow^{+} A_1^{\star}$' $\rightarrow^{+} A_2^{\star}$' $\rightarrow^{+} A_3^{\star}$' $\ldots \Rightarrow A^{\star}$' is not strongly normalizing in $\lambda^{\rightarrow}$. But this contradicts Theorem 8.3.9, so we must have that A is strongly normalizing under $\beta\eta$-reduction as desired. □

In Figure 8.1, I give the code for an algorithm to $\beta\eta$-normalize valid constructors based on this result. I have written the algorithm in functional pseudo-code. My version of pseudo-code allows raising and handling the single exception `fail`. Functions written in it can thus return a value, raise `fail`, or loop forever (functions can be defined recursively). Patterns in case expressions are matched sequentially in the order written, stopping with the first match; if no match is found, `fail` is raised. I allow guard conditions on the variables involved in a pattern in order to handle equivalence via α-conversion. Attempting to evaluate an undefined selection (e.g., $\mathcal{S}(<K>, x, .1)$) or an undefined assignment lookup (i.e., , (x) for $x \notin \text{dom}($, $)$) also results in `fail` being raised. I shall be using this notation throughout the dissertation.

The algorithm is composed of two procedures, *RO* and *BR*. *RO* is a reduction procedure, reducing its argument constructor by one $\beta\eta$-step. I have constructed *RO* so that it will only return its constructor unchanged if it is in $\beta\eta$-normal form. *BR* is the actual $\beta\eta$-normalizing procedure; it just applies *RO* iteratively to its argument until it stops changing. Theorem 8.3.16 ensures that *BR* always terminates on valid constructors. The correctness proofs for these algorithms are as follows:

Lemma [Tagged] 8.3.17 (Properties of *RO* algorithm I)

1. *RO*(A) *always returns*
2. $A \rightarrow_{\beta\eta}$ *RO*(A)

Proof: The first part follows from the fact that all recursive calls to *RO* are on smaller constructors and there are no points where fail can be raised. The second part is proved by structural induction on A. Example cases:

Lam: Here $A = \lambda\alpha{::}K.\,A_1$. Applying the induction hypothesis gives us that $A_1 \rightarrow_{\beta\eta}$ *RO*(A_1). By R-LAM then, $\lambda\alpha{::}K.\,A_1 \rightarrow_{\beta\eta} \lambda\alpha{::}K.\,RO(A_1)$. If $A_1 = A'\,\alpha$, $\alpha \notin$ FCV(A') then $\lambda\alpha{::}K.\,A_1 \rightarrow_{\beta\eta} A'$ via R-ETA and R-REFL. Hence, by inspection of *RO*, regardless of which path is taken, $A \rightarrow_{\beta\eta}$ *RO*(A).

```
$RO(A)$ = case $A$ of
          $\lambda\alpha{::}K.\,A_1$: let $A_1' = RO(A_1)$ in
                    if $A_1 \neq A_1'$ then
                      return$(\lambda\alpha{::}K.\,A_1')$
                    else
                      case $A_1$ of
              $(A'\,\alpha)$, $\alpha \notin \mathrm{FCV}(A')$: return$(A')$
                                  $A'$: return$(\lambda\alpha{::}K.\,A_1)$
                      end

             $A_1\,A_2$: let $A_1' = RO(A_1)$;
                      $A_2' = RO(A_2)$ in
                      if $A_1\,A_2 \neq A_1'\,A_2'$ then
                        return$(A_1'\,A_2')$
                      else
                        case $A_1$ of
                          $\lambda\alpha{::}K.\,A'$: return$([A_2/\alpha]A')$
                                 $A'$: return$(A_1\,A_2)$

          $\Pi x{:}A_1.\,A_2$: return$(\Pi x{:}RO(A_1).\,RO(A_2))$
          $\Sigma x{:}A_1.\,A_2$: return$(\Sigma x{:}RO(A_1).\,RO(A_2))$
         $<=A'{::}K>$: return$(<=RO(A'){::}K>)$
            $x\rho_{A'}\rho'!$: return$(x\rho_{RO(A')}\rho'!)$
                  $A'$: return$(A')$

$BR(A)$ = let $A' = RO(A)$ in
           if $A = A'$ then
             return$(A)$
           else
             return$(BR(A'))$
```

Figure 8.1: β-reducing a term

App: Here $A = A_1\, A_2$. Applying the induction hypothesis gives us that $A_1 \rightarrow_{\beta\eta} RO(A_1)$ and $A_2 \rightarrow_{\beta\eta} RO(A_2)$. By R-APP then, $A_1\, A_2 \rightarrow_{\beta\eta} RO(A_1)\, RO(A_2)$. If $A_1 = \lambda\alpha{::}K.\, A'$ then $A_1\, A_2 \rightarrow_{\beta\eta} [A_2/\alpha]A'$ via R-BETA and R-REFL. Hence, by inspection of RO, regardless of which path is taken, $A \rightarrow_{\beta\eta} RO(A)$.

□

Lemma [Tagged] 8.3.18 (Properties of RO algorithm II)
If $A = RO(A)$ then A is in $\beta\eta$-normal form.

Proof: We prove the contrapositive (if $\exists A'.\ A \rightarrow^1_{\beta\eta} A'$ then $A \neq RO(A)$) by structural induction on A. Example cases:

Lam: Given $\lambda\alpha{::}K.\, A_1 \rightarrow^1_{\beta\eta} A'$. Inspecting the rewriting rules shows that this implies that one of the following is true:

- $A_1 \rightarrow^1_{\beta\eta} A'_1$ for some A'_1
  $\Rightarrow A_1 \neq RO(A_1)$ (induction hypothesis)
  $\Rightarrow RO(\lambda\alpha{::}K.\, A_1) = \lambda\alpha{::}K.\, RO(A_1) \neq \lambda\alpha{::}K.\, A_1$
- A_1 is in $\beta\eta$-normal form and is equal to $A'\,\alpha$, $\alpha \notin \mathrm{FCV}(A')$.
  $\Rightarrow A_1 = RO(A_1)$ (Lemma 8.3.17 plus definition of normal form)
  $\Rightarrow RO(\lambda\alpha{::}K.\, A_1) = A' \neq \lambda\alpha{::}K.\, A_1 = \lambda\alpha{::}K.\, A'\,\alpha$

App: Given $A_1\, A_2 \rightarrow^1_{\beta\eta} A'$. Inspecting the rewriting rules shows that this implies that one of the following is true:

- $A_i \rightarrow^1_{\beta\eta} A'_i$ for some A'_i, $i \in \{1, 2\}$
  $\Rightarrow A_i \neq RO(A_i)$ (induction hypothesis)
  $\Rightarrow A_1\, A_2 \neq RO(A_1)\, RO(A_2)$
  $\Rightarrow RO(A_1\, A_2) = RO(A_1)\, RO(A_2) \neq A_1\, A_2$
- A_1 is in $\beta\eta$-normal form and has the form $\lambda\alpha{::}K.\, A''$, A_2 is also in $\beta\eta$-normal form, and $A' = [A_2/\alpha]A''$.
  $\Rightarrow A_1\, A_2 = RO(A_1)\, RO(A_2)$ (Lemma 8.3.17 plus definition of normal form)
  $\Rightarrow RO(A_1\, A_2) = [A_2/\alpha]A'' = A'$
  $\Rightarrow A \neq A' = RO(A)$ (definition of $\rightarrow^1_{\beta\eta}$)

□

Corollary [Tagged] 8.3.19 (Properties of BR algorithm)
If , $\vdash A :: K$ then

1. *$BR(A)$ always returns*
2. *$A \rightarrow^*_{\beta\eta} BR(A)$*
3. *$BR(A)$ is in $\beta\eta$-normal form.*

Proof: Inspection of the BR procedure reveals that: when called with A, it recurses on $RO(A)$, then $RO(RO(A)) = RO^2(A)$, then $RO^3(A)$, ..., then $RO^n(A)$ where $0 \leq n \leq \infty$ and $RO^i(A) \neq RO^{i+1}(A)$ for $0 \leq i < n$. If $BR(A)$ returns then it returns $RO^n(A)$, $n < \infty$, and $RO^n(A) = RO^{n+1}(A)$.

By Lemma 8.3.17, $A \rightarrow^1_{\beta\eta} RO(A) \rightarrow^1_{\beta\eta} RO^2(A) \ldots \rightarrow^1_{\beta\eta} RO^n(A)$. By Theorem 8.3.16 then, n must be finite. (Otherwise, A is not strongly normalizing under $\beta\eta$-reduction.) That $BR(A)$ always returns then follows since RO is known to always return by Lemma 8.3.17 and the above argument shows that BR calls itself only a finite number of times. From the transitivity of $\rightarrow^*_{\beta\eta}$, we have that $A \rightarrow^*_{\beta\eta} RO^n(A) \Rightarrow A \rightarrow^*_{\beta\eta} BR(A)$. Finally, by Lemma 8.3.18, $BR(A) = RO^n(A)$ is in $\beta\eta$-normal form. □

8.4 Canonical Form

In this section I define γ-final and canonical form and prove some useful properties about them. I start out with γ-final form:

Definition [Tagged] 8.4.1 (γ-final form)
A is in γ-final form iff $A = C[x\rho_{A'}\rho'!]$ implies that $\rho' = \epsilon$ and $A' = \ <K>$ for some K.

Lemma [Tagged] 8.4.2 *If A is in γ-final form then A is in γ-normal form.*

Proof: Suppose $A \rightarrow_\gamma A'$. We can show by structural induction on the derivation that $A = A'$ (example cases below). Hence, A is in γ-normal form.

EXT: Given $x\rho_{A_1}\rho'\rho''! \rightarrow_\gamma x\rho\rho'_{A_2}\rho''!$ derived via rule R-EXT from $A_1 \rightarrow_\gamma A'_1$ and $\mathcal{S}(A'_1, x\rho, \rho') = A_2$. By the definition of γ-final form, $\rho' = \rho'' = \epsilon$ and A_1 is in γ-final form $\Rightarrow A'_1 = A_2$. Applying the inductive hypothesis gives us that $A_1 = A'_1$ $\Rightarrow x\rho_{A_1}\rho'\rho''! = x\rho\rho'_{A_2}\rho''!$.

ABBREV: Given $x\rho_{A_1}! \rightarrow_\gamma A'_1$ derived via rule R-ABBREV from $A_1 \rightarrow_\gamma \ <=A'_1::K'>$. By the definition of γ-final form, $A_1 = \ <K>$ for some K. By shape preservation (Lemma 7.3.7), $<K> \rightarrow \ <=A'_1::K'>$ is impossible. Hence this case can't occur.

□

Lemma [Tagged] 8.4.3 *If A_1 and A_2 are in γ-final form then $[A_2/\alpha]A_1$ is in γ-final form.*

Proof: By structural induction on A_1. Example cases:

Var I: Here $A_1 = \alpha \Rightarrow [A_2/\alpha]A_1 = A_2$.

Ext: Here $A_1 = x\rho_{<K>}! \Rightarrow [A_2/\alpha]A_1 = A_1$, which is in γ-final form.

□

Lemma [Tagged] 8.4.4 *If A is in γ-final form and $A \rightarrow_{\beta\eta} A'$ then A' is in γ-final form.*

Proof: By structural induction on the derivation of $A \rightarrow_{\beta\eta} A'$. Example cases:

R-BETA: Given $(\lambda\alpha{::}K.\, A_1)\, A_2 \rightarrow_{\beta\eta} [A_2'/\alpha]A_1'$ derived via rule R-BETA from $A_1 \rightarrow_{\beta\eta} A_1'$ and $A_2 \rightarrow_{\beta\eta} A_2'$. By the definition of γ-final form, A_1 and A_2 are in γ-final form. Applying the induction hypothesis gives us that A_1' and A_2' are in γ-final form. Hence, by Lemma 8.4.3, $[A_2'/\alpha]A_1'$ is in γ-final form.

R-ETA: Given $\lambda\alpha{::}K.\, A\,\alpha \rightarrow_{\beta\eta} A'$ derived via rule R-ETA from $A \rightarrow_{\beta\eta} A'$ and $\alpha \notin \mathrm{FCV}(A)$. By the definition of γ-final form, A is in γ-final form. Applying the induction hypothesis gives us that A' is in γ-final form.

R-EXT: Given $x\rho_{A_1}\rho'\rho''! \rightarrow_{\beta\eta} x\rho\rho'_{A_2}\rho''!$, $\rho' = \epsilon$, derived via rule R-EXT from $A_1 \rightarrow_{\beta\eta} A_1'$ and $\mathcal{S}(A_1', x\rho, \rho') = A_2 \Rightarrow A_1' = A_2$. By the definition of γ-final form, $\rho'' = \epsilon$ and $A_1 = {<}K{>}$ for some K. By Lemma 7.3.7, $A_2 = {<}K{>} \Rightarrow A = A'$.

□

Here I define canonical form and show how new constructors in canonical form can be created from existing constructors in canonical form:

Definition [Tagged] 8.4.5 (Canonical form)
A is in canonical form iff A is in both $\beta\eta$-normal form and γ-final form.

Lemma [Tagged] 8.4.6 *Suppose A is in canonical form. Then:*

1. *$[x\rho/x']A$ is in canonical form.*

2. *If $\mathcal{S}(A, x\rho, \rho')$ exists then $\mathcal{S}(A, x\rho, \rho')$ is in canonical form.*

Proof: Proved sequentially by structural induction on A and ρ' respectively. $\square$

Lemma [Tagged] 8.4.7

1. *If* $, ,\alpha{::}K \vdash A :: K'$ *and* A *is in canonical form then* $BR(\lambda\alpha{::}K.\, A)$ *returns* A' *such that* $\lambda\alpha{::}K.\, A \rightarrow^* A'$ *and* A' *is in canonical form.*

2. *If* $, \vdash A_1 :: K_2{\Rightarrow}K$*,* $, \vdash A_2 :: K_2$*, and* A_1 *and* A_2 *are in canonical form then* $BR(A_1\, A_2)$ *returns* A' *such that* $A_1\, A_2 \rightarrow^* A'$ *and* A' *is in canonical form.*

Proof: The proof is similar for both parts; we give only the proof of the first part here:

By C-LAM, $, \vdash \lambda\alpha{::}K.\, A :: K{\Rightarrow}K'$. By the definition of canonical form, A is in γ-final form $\Rightarrow$ $\lambda\alpha{::}K.\, A$ is in γ-final form (by defn.). By Corollary 8.3.19, $BR(\lambda\alpha{::}K.\, A)$ always terminates, $\lambda\alpha{::}K.\, A \rightarrow^*_{\beta\eta} BR(\lambda\alpha{::}K.\, A)$, and $BR(\lambda\alpha{::}K.\, A)$ is in $\beta\eta$-normal form. By Lemma 8.4.4, $BR(\lambda\alpha{::}K.\, A)$ is in γ-final form $\Rightarrow$ $BR(\lambda\alpha{::}K.\, A)$ is in canonical form (by defn.). $\square$

Because canonical forms are normal forms, they are reducts of all constructors they are equal to. Since reducing a constructor preserves many properties about it (e.g., possession of certain outer shapes), this fact means that if a canonical form is equal to a constructor with certain properties then that canonical form must also have those properties:

Lemma [Tagged] 8.4.8 *If* A *is in canonical form then* A *is in normal form.*

Proof: By Lemma 8.4.2, A is in γ-normal form. Hence, by Corollary 8.1.5, A is in normal form. $\square$

Lemma [Tagged] 8.4.9 *If* $, \vdash A_1 = A_2 :: K$ *and* A_2 *is in canonical form then* $A_1 \rightarrow^* A_2$*.*

Proof: By Lemma 8.4.8, A_2 is in normal form. By Theorem 7.8.5, $A_1 \downarrow^* A_2 \Rightarrow \exists A.\ A_1 \rightarrow^* A$ and $A_2 \rightarrow^* A$. By the definition of normal form, $A = A_2$. Hence, $A_1 \rightarrow^* A_2$. $\square$

Lemma [Tagged] 8.4.10 (Shape Pullback)
Suppose $, \vdash A_1 = A_2 :: K$ *and* A_1 *is in canonical form. Then:*

1. *If* $A_2 = \alpha$, $A_2 = \mathbf{ref}$, $A_2 = \mathbf{rec}$, *or* $A_2 = {<}K{>}$ *then* $A_1 = A_2$.
2. *If* $A_2 = \Pi x{:}A_2'.\, A_2''$ *then* $\exists A_1', A_1''.\ A_1 = \Pi x{:}A_1'.\, A_1''$.
3. *If* $A_2 = \Sigma x{:}A_2'.\, A_2''$ *then* $\exists A_1', A_1''.\ A_1 = \Sigma x{:}A_1'.\, A_1''$.
4. *If* $A_2 = {<}{=}A_2'{::}K{>}$ *then* $\exists A_1'.\ A_1 = {<}{=}A_1'{::}K{>}$.
5. *If* $A_2 = \mathbf{ref}\, A_2'$ *then* $\exists A_1'.\ A_1 = \mathbf{ref}\, A_1'$.
6. *If* $A_2 = \mathbf{rec}\, A_2'$ *then* $\exists A_1'.\ A_1 = \mathbf{rec}\, A_1'$.

Proof: By Lemma 8.4.9 and E-REFL, $A_2 \rightarrow^* A_1$. The desired results then follow from Lemma 7.3.7. □

Corollary [Tagged] 8.4.11 (Canonical Strengthening)
If $, _1, x{:}A; , _2 \vdash A_1 = A_2 :: K$, A_1 *is in canonical form, and* $, _1; , _2 \vdash A_2 :: K$ *then* $, _1; , _2 \vdash A_1 = A_2 :: K$.

Proof: By E-SYM and Lemma 8.4.9, $A_2 \rightarrow^* A_1 \Rightarrow\ , _1; , _2 \vdash A_1 :: K$ (Corollary 7.6.17) $\Rightarrow\ , _1; , _2 \vdash A_1 = A_2 :: K$ (Corollary 7.8.6). □

By a similar argument using Parallelism II (Theorem 7.3.13), we can "pull back" selections through equality to a canonical form; by doing this repeatedly, we can reduce place lookup using an assignment in canonical form to checking equality:

Lemma [Tagged] 8.4.12 (Pullback)
If A *is in canonical form,* $, \vdash A = A' :: \Omega$, *and* $, \vdash \mathcal{S}(A', x\rho, \rho') :: \Omega$ *then* $, \vdash \mathcal{S}(A, x\rho, \rho') = \mathcal{S}(A', x\rho, \rho') :: \Omega$.

Proof: By E-SYM and Lemma 8.4.9, $A' \rightarrow^* A \Rightarrow \mathcal{S}(A', x\rho, \rho') \rightarrow^* \mathcal{S}(A, x\rho, \rho')$ (Theorem 7.3.13) $\Rightarrow\ , \vdash \mathcal{S}(A, x\rho, \rho') :: \Omega$ (Corollary 7.6.17) $\Rightarrow\ , \vdash \mathcal{S}(A, x\rho, \rho') = \mathcal{S}(A', x\rho, \rho') :: \Omega$ (Theorem 7.8.5). □

Lemma [Tagged] 8.4.13 *If* $, (x)$ *is in canonical form and* $, \vdash x\rho \Rightarrow A$ *then* $, \vdash A = \mathcal{S}(, (x), x, \rho) :: \Omega$ *and* $\mathcal{S}(, (x), x, \rho)$ *is in canonical form.*

Proof: By structural induction on the derivation of $, \vdash x\rho \Rightarrow A$. Example cases:

P-INIT: Given $, \vdash x \Rightarrow A$ ($\rho = \epsilon$) derived via rule P-INIT from $\vdash ,$ valid and $x{:}A \in ,$ $\Rightarrow \mathcal{S}(, (x), x, \rho) = , (x)$. By Lemma 6.8.11, $, (x) = A$ and $, \vdash A :: \Omega$. Hence, by E-REFL, $, \vdash A = \mathcal{S}(, (x), x, \rho) :: \Omega$.

P-MOVE: Given , $\vdash x\rho\rho' \Rightarrow A$ derived via rule P-MOVE from , $\vdash x\rho \Rightarrow A'$, , $\vdash A' = A'' :: \Omega$, and $\mathcal{S}(A'', x\rho, \rho') = A$. Applying the induction hypothesis then gives us that , $\vdash A' = \mathcal{S}(,\,(x), x, \rho) :: \Omega$ and $\mathcal{S}(,\,(x), x, \rho)$ is in canonical form $\Rightarrow$, $\vdash \mathcal{S}(,\,(x), x, \rho) = A'' :: \Omega$ (E-TRAN and E-SYM). By Theorem 7.6.14, , $\vdash A :: \Omega$. Hence, by Lemma 8.4.12, , $\vdash \mathcal{S}(\mathcal{S}(,\,(x), x, \rho), x\rho, \rho') = A :: \Omega$ $\Rightarrow$, $\vdash A = \mathcal{S}(,\,(x), x, \rho\rho') :: \Omega$ (Lemma 6.8.1 and E-SYM) By Lemma 8.4.6, $\mathcal{S}(,\,(x), x, \rho\rho')$ is in canonical form.

□

8.5 Assignment Reduction

For efficiency reasons, the normalization and validity checking algorithm I give only reduces the assignment once rather than reducing parts of it each time they are needed. In order to do this, I shall need to extend the notions of reduction and canonical form to assignments:

Definition [Tagged] 8.5.1 (Reduction of assignments) *Reduction can be extended to assignments as follows:*

$$\bullet \rightarrow \bullet \qquad \text{(R-EMPTY)}$$

$$\frac{,\ \rightarrow\ ,'}{,\ ,\alpha{::}K \rightarrow\ ,', \alpha{::}K} \qquad \text{(R-DECL-C)}$$

$$\frac{,\ \rightarrow\ ,' \qquad A \rightarrow A'}{,\ , x{:}A \rightarrow\ ,', x{:}A'} \qquad \text{(R-DECL-T)}$$

Definition [Tagged] 8.5.2 (Canonical form for assignments)
, *is in canonical form iff* $\forall x{:}A \in\ ,\,.\ A$ *is in canonical form.*

I shall need that reducing the assignments in judgments leaves the judgments' truth values unchanged and that reduction on assignments preserves assignment validity:

Lemma [Tagged] 8.5.3

1. *If* $,_1;\, ,_2 \rightarrow\ ,'$ *then* $\exists\, ,'_1, ,'_2$ *such that* $,' =\ ,'_1;\, ,'_2$, $,_1 \rightarrow\ ,'_1$, *and* $,_2 \rightarrow\ ,'_2$.
2. *If* $,' \rightarrow\ ,_1;\, ,_2$ *then* $\exists\, ,'_1, ,'_2$ *such that* $,' =\ ,'_1;\, ,'_2$, $,'_1 \rightarrow\ ,_1$, *and* $,'_2 \rightarrow\ ,_2$.

Proof: By structural induction on , $_2$. □

Lemma [Tagged] 8.5.4 *Suppose* , $\rightleftharpoons$, $'$. *Then:*

1. *dom*(,) = *dom*(, $'$)
2. *If* $\alpha{::}K \in$, *then* $\alpha{::}K \in$, $'$.

Lemma [Tagged] 8.5.5 (Reduction and assignments)
Suppose $\vdash$, *valid,* $\vdash$, $'$ *valid, and* , $\to$, $'$. *Then:*

1. $\vdash$, , D *valid iff* $\vdash$, $'$, D *valid.*
2. , $\vdash A :: K$ *iff* , $' \vdash A :: K$.
3. , $\vdash x\rho \Rightarrow A$ *iff* , $' \vdash x\rho \Rightarrow A$.
4. , $\vdash A_1 = A_2 :: K$ *iff* , $' \vdash A_1 = A_2 :: K$.

Proof: We prove an equivalent version of the lemma where the rewriting precondition is instead that , $\rightleftharpoons$, $'$ and the results are that the judgments involving , imply the judgments involving , $'$. We prove this by simultaneous structural induction on the derivations. Interesting cases:

DECL-T: Given $\vdash$, , $x{:}A$ valid derived via rule DECL-T from , $\vdash A :: \Omega$ and $x \notin$ dom(,). Applying the induction hypothesis gives us that , $' \vdash A :: \Omega$ By Lemma 8.5.4, dom(,) = dom(, $'$) $\Rightarrow x \notin$ dom(, $'$). Hence, by DECL-T, $\vdash$, $'$, $x{:}A$ valid.

C-VAR: Given , $\vdash \alpha :: K$ derived via rule C-VAR from $\vdash$, valid and $\alpha{::}K \in$, $\Rightarrow \alpha{::}K \in$, $'$ (Lemma 8.5.4) $\Rightarrow$, $' \vdash \alpha :: K$ (C-VAR).

C-DFUN: Given , $\vdash \Pi x{:}A.\, A' :: \Omega$ derived via rule C-DFUN from , , $x{:}A \vdash A' :: \Omega$ $\Rightarrow\ \vdash$, , $x{:}A$ valid (Theorem 6.8.3). Applying the induction hypothesis gives us that $\vdash$, $'$, $x{:}A$ valid. By R-DECL-T and R-REFL, , , $x{:}A \rightleftharpoons$, $'$, $x{:}A$. Applying the induction hypothesis again gives us that , $'$, $x{:}A \vdash A' :: \Omega \Rightarrow$, $' \vdash \Pi x{:}A.\, A' :: \Omega$ (C-DFUN).

P-INIT: Given , $\vdash x \Rightarrow A$ derived via rule P-INIT from $\vdash$, valid and $x{:}A \in$, $\Rightarrow \exists$, $_1$, , $_2$. , = (, $_1$, $x{:}A$; , $_2$). $\Rightarrow\ \vdash$, $_1$, $x{:}A$ valid and $\vdash$, $_1$ valid are sub-derivations of , $\vdash x \Rightarrow A$ (Theorem 6.8.3) $\Rightarrow$, $_1 \vdash A :: \Omega$ is a sub-derivation of , $\vdash x \Rightarrow A$ (DECL-T). By Lemma 8.5.3 and R-DECL-T, $\exists$, $'_1$, A', , $'_2$ such that , $'$ = , $'_1$, $x{:}A'$; , $'_2$, , $_1 \rightleftharpoons$, $'_1$, and $A \rightleftharpoons A'$. $\Rightarrow\ \vdash$, $'_1$ valid (Theorem 6.8.3).

Applying the induction hypothesis gives us that , $'_1 \vdash A :: \Omega \Rightarrow$, $' \vdash A :: \Omega$ (Theorem 6.8.10). By P-INIT, , $' \vdash x \Rightarrow A'$. By Lemma 6.8.11, , $' \vdash A' :: \Omega$. Hence, by Theorem 7.8.5, , $' \vdash A' = A :: \Omega \Rightarrow$, $' \vdash x \Rightarrow A$ (P-MOVE).

□

Theorem [Tagged] 8.5.6 (Subject reduction for assignments)
If $\vdash$, *valid and* , $\rightarrow$, $'$ *then* $\vdash$, $'$ *valid.*

Proof: By structural induction on the derivation of $\vdash$, valid. The interesting case is as follows:

DECL-T: Given $\vdash$, , $x{:}A$ valid derived via rule DECL-T from , $\vdash A :: \Omega$ and $x \notin$ dom(,) as well as , , $x{:}A \rightarrow$, $'$, $x{:}A'$ derived via rule R-DECL-T from , $\rightarrow$, $'$ and $A \rightarrow A'$ $\Rightarrow\ \vdash$, valid (Theorem 6.8.3). Applying the induction hypothesis gives us that $\vdash$, $'$ valid. By Lemma 8.5.5, , $' \vdash A :: \Omega$. By subject reduction (Corollary 7.6.17), , $' \vdash A' :: \Omega$. By Lemma 8.5.4, dom(,) = dom(, $'$) $\Rightarrow x \notin$ dom(, $'$). Hence, by DECL-T, $\vdash$, $'$, $x{:}A'$ valid.

□

8.6 Decidability

In this section I give and prove correct algorithms for use in deciding all the tagged judgments. For each judgment, I give an algorithm that decides that judgment given certain preconditions about its arguments (e.g., the assignment is already normalized in constructor validity case). Later on I give versions based on the original algorithms that have no preconditions. This organization allows extra work such as repeatedly reducing assignments to be avoided.

In Figure 8.2, I give the code for the algorithms for the equal constructors (EC') and place lookup (PL') judgments. EC' takes as input two pairs, each containing one constructor in canonical form and the kind of that constructor relative to an (unpassed) assignment , . If the two constructors are equal under , then EC' returns their kind. Otherwise, it raises `fail`. Because the constructors are in normal form, it can do this just by comparing them and their kinds.

Lemma [Tagged] 8.6.1 (Properties of EC' algorithm)

1. $EC'((A_1, K_1), (A_2, K_2))$ *always terminates.*

2. *If* , $\vdash A_1 :: K_1$, , $\vdash A_2 :: K_2$, *and* A_1 *and* A_2 *are in canonical form then* $EC'((A_1, K_1), (A_2, K_2))$ *returns* K *iff* , $\vdash A_1 = A_2 :: K$.

```
$EC'((A_1, K_1), (A_2, K_2))$ = if $K_1 = K_2$ and $A_1 = A_2$ then
                                    return($K_1$)
                                else
                                    raise fail

$PL'(, , x\rho, A) = EC'((\mathcal{S}(, (x), x, \rho), \Omega), (A, \Omega))$

$VT'(, , A)$ = case $VC'(, , A)$ of $(A', \Omega)$: return($A'$)
```

Figure 8.2: Determining the equality and place lookup judgments

Proof: The first part follows from inspection of the EC' procedure. The second part is proved as follows:

By Lemma 8.4.8, A_1 and A_2 are in normal form. Hence, by Lemma 8.1.3, $A_1 \downarrow^* A_2$ iff $A_1 = A_2$.

$\Rightarrow$) By inspection of the EC' procedure, $EC'((A_1, K_1), (A_2, K_2))$ returns K implies that $K = K_1 = K_2$ and $A_1 = A_2 \Rightarrow A_1 \downarrow^* A_2$. Hence, by Theorem 7.8.5, $, \vdash A_1 = A_2 :: K$.

$\Leftarrow$) By Theorem 7.6.10, $, \vdash A_1 :: K$ and $, \vdash A_2 :: K$. By Theorem 7.6.11, $K_1 = K_2 = K$. By Theorem 7.8.5, $A_1 \downarrow^* A_2 \Rightarrow A_1 = A_2$ $\Rightarrow EC'((A_1, K_1), (A_2, K_2))$ returns K.

□

If $, \vdash A :: \Omega$ and , and A are both in canonical form, then $PL'(, , x\rho, A)$ returns iff $, \vdash x\rho \Rightarrow A$. It does this by using Lemma 8.4.13 to reduce place lookup to equality checking and then using EC' to decide the resulting equality question.

Lemma [Tagged] 8.6.2 (Properties of PL' algorithm)

1. *$PL'(, , x\rho, A)$ always terminates.*
2. *If $, \vdash A :: \Omega$ and , and A in canonical form then $PL'(, , x\rho, A)$ returns iff $, \vdash x\rho \Rightarrow A$.*

Proof: The first part follows from inspection of the PL' procedure plus Lemma 8.6.1. The second part is proved as follows:

$\Rightarrow$) By inspection of the PL' algorithm, $PL'(,\ , x\rho,\ A)$ returns implies that $EC'((\mathcal{S}(,\ (x), x, \rho), \Omega),\ (A, \Omega))$ returns $\Rightarrow$, (x) and $\mathcal{S}(,\ (x), x, \rho)$ exist $\Rightarrow x \in \mathrm{dom}(,\ )$. By inspection of the EC' algorithm, we also have that $\mathcal{S}(,\ (x), x, \rho) = A$. By Lemma 6.8.3, $\vdash$, valid. By Theorem 6.8.11, , $\vdash$, $(x) :: \Omega$. By P-INIT then, , $\vdash x \Rightarrow$, (x). By P-MOVE and E-REFL then, , $\vdash x\rho \Rightarrow \mathcal{S}(,\ (x), x, \rho)$. Hence, , $\vdash x\rho \Rightarrow A$.

$\Leftarrow$) By Theorem 6.8.5, $x \in \mathrm{dom}(,\ ) \Rightarrow$, (x) exists. By the definition of canonical form, , (x) is in canonical form. Hence, by Lemma 8.4.13, , $\vdash A = \mathcal{S}(,\ (x), x, \rho) :: \Omega$ and $\mathcal{S}(,\ (x), x, \rho)$ is in canonical form. By Theorem 7.6.10, , $\vdash \mathcal{S}(,\ (x), x, \rho) :: \Omega$. By Lemma 8.6.1 then, $EC'((\mathcal{S}(,\ (x), x, \rho), \Omega),\ (A, \Omega))$ returns $\Omega \Rightarrow PL'(,\ , x\rho,\ A)$ returns.

□

Also in Figure 8.2 is the code for an algorithm for checking *type* validity (VT'). It is defined using the algorithm for checking general constructor validity (VC'), the code for which may be found in Figure 8.3. These algorithms are mutually recursive and differ mainly in their return behavior. Both take as arguments a valid assignment in canonical form and a constructor. They check the validity of the constructor under the given assignment while normalizing the constructor. If the constructor is invalid or, in the case of VT', has kind other than Ω, then `fail` is raised. Otherwise, a canonical form for the constructor is returned; VC' returns the constructor's kind under the given assignment as well in a pair with the canonical form. These algorithms implement the ideas discussed in Section 8.2.

Lemma [Tagged] 8.6.3 (Properties of VC' and VT' algorithms I)
If $\vdash$, *valid and* , *is in canonical form then:*

1. $VT'(,\ , A)$ *and* $VC'(,\ , A)$ *alway terminate.*

2. *If* $VT'(,\ , A)$ *returns* A' *then* , $\vdash A :: \Omega$, $A \rightarrow^* A'$, *and* A' *is in canonical form.*

3. *If* $VC'(,\ , A)$ *returns* (A', K) *then* , $\vdash A :: K$, $A \rightarrow^* A'$, *and* A' *is in canonical form.*

Proof: All parts are proved simultaneously by induction on the procedure call in question under the following metric:

$$\begin{array}{rcl} |VT'(,\ , A)| & = & 2|A| + 1 \\ |VC'(,\ , A)| & = & 2|A| \end{array}$$

```
$VC'(, ,A)$ = case $A$ of
                         $\alpha$: case , of
                                     $(,_1, \alpha{::}K; ,_2)$, $\alpha \notin$ dom$(,_1)$: return$(\alpha, K)$
  $\Pi x{:}A_1.\, A_2$, $x \notin$ dom$(, )$: let $A_1' = VT'(, , A_1)$ in
                                     return$(\Pi x{:}A_1'.\, VT'((, , x{:}A_1'), A_2), \Omega)$
                                 end
  $\Sigma x{:}A_1.\, A_2$, $x \notin$ dom$(, )$: let $A_1' = VT'(, , A_1)$ in
                                     return$(\Sigma x{:}A_1'.\, VT'((, , x{:}A_1'), A_2), \Omega)$
                                 end
   $\lambda\alpha{::}K.\, A$, $\alpha \notin$ dom$(, )$: let $(A', K') = VC'((, , \alpha{::}K), A)$ in
                                     return$(BR(\lambda\alpha{::}K.\, A'), K{\Rightarrow}K')$
                                 end
                     $A_1\, A_2$: let $(A_1', K_1') = VC'(, , A_1)$
                                     $(A_2', K_2') = VC'(, , A_2)$
                                     $(K{\Rightarrow}K') = K_1'$ in
                                         if $K = K_2'$ then
                                             return$(BR(A_1'\, A_2'), K')$
                                         else
                                             raise fail
                                 end
              $<=A{::}K>$: let $(A', K') = VC'(, , A)$ in
                                     if $K = K'$ then
                                         return$(<=A'{::}K>, \Omega)$
                                     else
                                         raise fail
                 $x\rho_{1_A}\rho_2!$: let $A' = \mathcal{S}(, (x), x, \rho_1\rho_2)$ in
                                         $PL'(, , x\rho_1, VT'(, , A))$;
                                         case $A'$ of
                                                 $<K>$: return$(x\rho_1\rho_{2<K>}!, K)$
                                           $<=A''{::}K>$: return$(A'', K)$
                                 end
                     $<K>$: return$(<K>, \Omega)$
                      **rec**: return(**rec**, $(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega$)
                      **ref**: return(**ref**, $\Omega{\Rightarrow}\Omega$)
```

Figure 8.3: Determining the validity of a constructor

where $|A|$ is any reasonable metric on constructors such that if A' is a proper sub-constructor of A then $0 \leq |A'| < |A|$. (That is to say, VT' may call VC' with the same constructor and VC' may call either procedure with a strictly smaller constructor.)

The parts involving VT' follow from the parts about VC' plus inspection of the VT' procedure. The interesting cases for VC' are as follows. Unless otherwise indicated, termination follows by inspection of the relevant code.

Var: Here $A = \alpha$. By inspection, if we return, we return $(\alpha,\, K)$ where $\alpha{::}K \in$, $\Rightarrow$, $\vdash \alpha :: K$ (C-VAR). By R-REFL, $\alpha \rightarrow^* \alpha$. By the definition of canonical form, α is in canonical form.

Dfun: Here $A = \Pi x{:}A_1.\, A_2$, $x \notin \mathrm{dom}(\,,\,)$. By the induction hypothesis, $VT'(\,,\,A_1)$ terminates; if it returns A_1' then , $\vdash A_1 :: \Omega$, $A_1 \rightarrow^* A_1'$, and A_1' is in canonical form $\Rightarrow$, , $x{:}A_1'$ is in canonical form (by defn.) and $\Pi x{:}A_1.\, A_2 \rightarrow^* \Pi x{:}A_1'.\, A_2$ (R-DFUN and R-REFL).

Assume it returns. By repeated use of subject reduction (Corollary 7.6.17), , $\vdash A_1' :: \Omega$ $\Rightarrow \vdash$, , $x{:}A_1'$ valid (DECL-T). By Theorem 7.8.5, , $\vdash A_1' = A_1 :: \Omega$. Applying the induction hypothesis again gives us that $VT'((\,,\,x{:}A_1'),\, A_2)$ terminates; if it returns A_2' then , , $x{:}A_1' \vdash A_2 :: \Omega$, $A_2 \rightarrow^* A_2'$, and A_2' is in canonical form $\Rightarrow \Pi x{:}A_1'.\, A_2'$ is in canonical form (by defn.) and $\Pi x{:}A_1'.\, A_2 \rightarrow^* \Pi x{:}A_1'.\, A_2'$ (R-DFUN and R-REFL) $\Rightarrow \Pi x{:}A_1.\, A_2 \rightarrow^* \Pi x{:}A_1'.\, A_2'$ (transitivity).

Assume it returns. By Theorem 7.6.9, , , $x{:}A_1 \vdash A_2 :: \Omega \Rightarrow$, $\vdash \Pi x{:}A_1.\, A_2 :: \Omega$ (C-DFUN). Thus, by inspection of the relevant code in VC', we must terminate and moreover, if we return, we return $(\Pi x{:}A_1'.\, A_2',\, \Omega)$ which has the needed properties.

Lam: Here $A = \lambda\alpha{::}K.\, A_1$, $\alpha \notin \mathrm{dom}(\,,\,)$. By DECL-C, $\vdash$, , $\alpha{::}K$ valid. By the definition of canonical form, , , $\alpha{::}K$ is in canonical form. By the induction hypothesis then, $VC'((\,,\,\alpha{::}K),\, A_1)$ terminates; if it returns (A_1', K') then , , $\alpha{::}K \vdash A_1 :: K'$, $A_1 \rightarrow^* A_1'$, and A_1' is in canonical form $\Rightarrow$, $\vdash \lambda\alpha{::}K.\, A_1 :: K{\Rightarrow}K'$ (C-LAM) and $\lambda\alpha{::}K.\, A_1 \rightarrow^* \lambda\alpha{::}K.\, A_1'$ (R-LAM).

Assume it returns. By repeated use of subject reduction (Corollary 7.6.17), , , $\alpha{::}K \vdash A_1' :: K'$, By Lemma 8.4.7, $BR(\lambda\alpha{::}K.\, A_1')$ returns A' such that $\lambda\alpha{::}K.\, A_1'$ $\rightarrow^* A'$ and A' is in canonical form $\Rightarrow \lambda\alpha{::}K.\, A_1 \rightarrow^* A'$ (transitivity). Thus, by inspection of the relevant code in VC', we must terminate and moreover, if we return, we return $(A',\, K{\Rightarrow}K')$ which has the needed properties.

App: Here $A = A_1\, A_2$. By the induction hypothesis, $VC'(\,,\,A_1)$ terminates; if it returns (A_1', K_1') then , $\vdash A_1 :: K_1'$, $A_1 \rightarrow^* A_1'$, and A_1' is in canonical form and $VC'(\,,\,A_2)$ terminates; if it returns (A_2', K_2') then , $\vdash A_2 :: K_2'$, $A_2 \rightarrow^* A_2'$, and A_2' is in canonical form $\Rightarrow A_1\, A_2 \rightarrow^* A_1'\, A_2'$ (R-APP).

Assume they both return. Then, by repeated use of subject reduction (Corollary 7.6.17), $, \vdash A_1' :: K_1'$ and $, \vdash A_2' :: K_2'$.

Assume $\exists K, K'$ such that $K_1' = K \Rightarrow K'$ and $K_2' = K$. Then, by C-APP, $, \vdash A_1\,A_2 :: K'$. By Lemma 8.4.7, $BR(A_1'\,A_2')$ returns A' such that $A_1'\,A_2' \rightarrow^* A'$ and A' is in canonical form $\Rightarrow A_1\,A_2 \rightarrow^* A'$ (transitivity). Thus, by inspection of the relevant code in VC', we must terminate and moreover, if we return, we return $(A',\ K')$ which has the needed properties.

Trans: Here $A = \,<\!=\!A_1{::}K\!>$. By the induction hypothesis, $VC'(,\ , A_1)$ terminates; if it returns (A_1', K_1') then $, \vdash A_1 :: K_1'$, $A_1 \rightarrow^* A_1'$, and A_1' is in canonical form $\Rightarrow$ $, \vdash \,<\!=\!A_1{::}K_1'\!> :: \Omega$ (C-TRANS), $<\!=\!A_1{::}K_1'\!> \rightarrow^* \,<\!=\!A_1'{::}K_1'\!>$ (R-TRANS), and $<\!=\!A_1'{::}K_1'\!>$ is in canonical form (by defn.).

Assume it returns and $K_1' = K$. Then we return $(<\!=\!A_1'{::}K\!>,\ \Omega)$ which has the needed properties. Otherwise, by inspection of the relevant code, we must terminate by failing.

Ext: Here $A = x\rho_{A_1}\rho'!$. By Lemma 8.6.2 and the induction hypothesis, $PL'(,\ , x\rho_1, VT'(,\ , A))$ always terminates $\Rightarrow$ we always terminate in this case.

If we are going to return, then $A' = \mathcal{S}(,\ (x), x, \rho_1\rho_2)$ must exist. By Lemma 8.4.6 and the fact that , is in canonical form, A' is in canonical form. By Theorem 6.8.11, $, \vdash\ , (x) :: \Omega$. By P-INIT then, $, \vdash x \Rightarrow\ , (x)$. By P-MOVE and E-REFL then, $, \vdash x\rho_1\rho_2 \Rightarrow A'$ Hence, by Theorem 7.6.14, $, \vdash A' :: \Omega$.

There are 2 cases where we return:

- Here $\exists K.\ A' = \,<\!K\!> \Rightarrow\ , \vdash x\rho_1\rho_{2_{<K>}}! :: K$ (C-EXT-O2). Since we return $(x\rho_1\rho_{2_{<K>}}!,\ K)$ for this case, the needed properties are satisfied.
- Here $\exists A'', K.\ A' = \,<\!=\!A''{::}K\!> \Rightarrow A''$ is in canonical form (by defn.) and $, \vdash A'' :: K$ (C-TRANS). Since we return $(A'',\ K)$ for this case, the needed properties are satisfied.

Rec: Here $A = \mathbf{rec}$. By C-REC, $, \vdash \mathbf{rec} :: (\Omega \Rightarrow \Omega) \Rightarrow \Omega$. By R-REFL, $\mathbf{rec} \rightarrow^* \mathbf{rec}$. By the definition of canonical form, **rec** is in canonical form. Inspection of the relevant code shows that $VC'(,\ , \mathbf{rec})$ always returns $(\mathbf{rec}, (\Omega \Rightarrow \Omega) \Rightarrow \Omega)$.

□

Lemma [Tagged] 8.6.4 (Properties of VC' and VT' algorithms II)
If , is in canonical form then:

1. *If* $, \vdash A :: \Omega$ *then* $VT'(, , A)$ *returns.*

2. *If* $, \vdash A :: K$ *then* $\exists A'.\ VC'(, , A)$ *returns* (A', K).

Proof: Both parts are proved simultaneously by induction on the procedure call in question using the same metric as in the previous lemma. The first part follows easily from induction using the second part and inspection of the definition of VT'. The interesting cases for the second part are as follows:

C-VAR: Given $, \vdash \alpha :: K$ derived via rule C-VAR from $\vdash ,$ valid and $\alpha{::}K \in ,\ \Rightarrow \exists ,_1, ,_2.$ $, = (,_1, \alpha{::}K; ,_2)$, $\alpha \notin \mathrm{dom}(,_1) \Rightarrow VC'(, , \alpha)$ returns (α, K).

C-DFUN: Given $, \vdash \Pi x{:}A_1.\, A_2 :: \Omega$, $x \notin \mathrm{dom}(,)$, derived via rule C-DFUN from $, , x{:}A_1 \vdash A_2 :: \Omega$ $\Rightarrow , \vdash A_1 :: \Omega$ (Theorem 6.8.3). Applying the induction hypothesis gives us that $VT'(, , A_1)$ returns. By Lemma 8.6.3, $A_1 \rightarrow^* VT'(, , A_1)$ and $VT'(, , A_1)$ is in canonical form $\Rightarrow , , x{:}VT'(, , A_1)$ is in canonical form (by defn.). By subject reduction (Corollary 7.6.17), $, \vdash VT'(, , A_1) :: \Omega$. By Theorem 7.8.5 then, $, \vdash A_1 = VT'(, , A_1) :: \Omega$. Hence, by Theorem 7.6.9, $, , x{:}VT'(, , A_1) \vdash A_2 :: \Omega$. Applying the induction hypothesis again gives us that $VT'((, , x{:}VT'(, , A_1)), A_2)$ returns. Hence, by inspection, $VC'(, , \Pi x{:}A_1.\, A_2)$ returns with a pair whose second element is Ω as required.

C-LAM: Given $, \vdash \lambda\alpha{::}K.\, A :: K{\Rightarrow}K'$, $\alpha \notin \mathrm{dom}(,)$, derived via rule C-LAM from $, , \alpha{::}K \vdash A :: K' \Rightarrow\ \vdash , , \alpha{::}K$ valid (Theorem 6.8.3). By the definition of canonical form, $, , \alpha{::}K$ is in canonical form. Applying the induction hypothesis then gives us that $\exists A'.\ VC'((, , \alpha{::}K), A)$ returns (A', K'). By Lemma 8.6.3 and subject reduction (Corollary 7.6.17), $, , \alpha{::}K \vdash A' :: K' \Rightarrow , \vdash \lambda\alpha{::}K.\, A' :: K{\Rightarrow}K'$ (C-LAM) $\Rightarrow BR(\lambda\alpha{::}K.\, A')$ returns (Lemma 8.3.19). Hence, by inspection of VC', $VC'(, , \lambda\alpha{::}K.\, A)$ returns with a pair whose second element is $K{\Rightarrow}K'$ as required.

C-EXT-O2: Given $, \vdash x\rho_{1\,A}\rho_2! :: K$ derived via rule C-EXT-O2 from $, \vdash x\rho_1\rho_2 \Rightarrow \,<\!K\!>$ and $, \vdash x\rho_1 \Rightarrow A$. By Theorem 7.6.15, $, \vdash A :: \Omega$. Applying the induction hypothesis gives us that $VT'(, , A)$ returns. By Lemma 8.6.3 and Theorem 6.8.3, $A \rightarrow^* VT'(, , A)$ and $VT'(, , A)$ is in canonical form. By subject reduction (Corollary 7.6.17), $, \vdash VT'(, , A) :: \Omega$. By Theorem 7.8.5, $, \vdash A = VT'(, , A) :: \Omega \Rightarrow , \vdash x\rho_1 \Rightarrow VT'(, , A)$ (P-MOVE). Hence, by Lemma 8.6.2, $PL'(, , x\rho_1, VT'(, , A))$ returns.

By Theorem 6.8.5, $x \in \mathrm{dom}(,) \Rightarrow , (x)$ is in canonical form (by defn.) Hence, by Lemma 8.4.13, $, \vdash \,<\!K\!> = \mathcal{S}(, (x), x, \rho_1\rho_2) :: \Omega$ and $\mathcal{S}(, (x), x, \rho_1\rho_2)$ is in canonical form $\Rightarrow \,<\!K\!> \rightarrow^* \mathcal{S}(, (x), x, \rho_1\rho_2)$ (Lemma 8.4.9) $\Rightarrow \mathcal{S}(, (x), x, \rho_1\rho_2) = \,<\!K\!>$ (Lemma 7.3.7). Hence, by inspection we see that in this case $VC'(, , x\rho_{1\,A}\rho_2!)$ returns a pair whose second element is K as required.

```
$VA'(,\ ) =$ case , of

              $\bullet$: return($\bullet$)

       $,\,',\alpha{::}K$: if $\alpha \in$ dom(, $'$) then
                   raise fail
               else
                   return($VA'(,\,')$,$\alpha{::}K$)

        $,\,',x{:}A$: let , $'' = VA'(,\,')$ in
                   if $x \in$ dom(, $'$) then
                       raise fail
                   else
                       return(, $''$,$x{:}VT'(,\,'',A)$)
               end
```

Figure 8.4: Determining the validity of an assignment

□

Figure 8.4 contains the code for the valid assignment judgment algorithm (VA'). It takes as an argument an assignment and returns a canonical form for that assignment if it is valid. Otherwise, it raises `fail`. The code is quite simple and uses VT'.

Lemma [Tagged] 8.6.5 (Properties of VA' algorithm I)

1. *$VA'(,\ )$ always terminates.*
2. *If $VA'(,\ )$ returns , $'$ then $\vdash$, valid, , $\rightarrow^*$, $'$, and , $'$ is in canonical form.*

Proof: By structural induction on , . Example case:

DECL-T: Here , = , $'$, $x{:}A$. Applying the induction hypothesis gives us that $VA'(,\,')$ always terminates and that if $VA'(,\,')$ returns , $''$ then $\vdash$, $'$ valid, , $'$ $\rightarrow^*$, $''$, and , $''$ is in canonical form.

Assume we return , $''$ and $x \notin$ dom(, $'$). By Theorem 8.5.6, $\vdash$, $''$ valid. Hence, by Lemma 8.6.3, $VT'(,\,'',A)$ always terminates and if $VT'(,\,'',A)$ returns A' then , $'' \vdash A :: \Omega$, $A \rightarrow^* A'$, and A' is in canonical form.

Assume $VT'(, '', A)$ returns A'. Hence, by R-DECL-T, $, ', x{:}A \rightarrow^* , '', x{:}A'$. By the definition of canonical form, $, '', x{:}A'$ is in canonical form. By Lemma 8.5.5, $, ' \vdash A :: \Omega \Rightarrow \vdash , ', x{:}A$ valid.

Thus, by inspection of the relevant code in VA', we must terminate and moreover, if we return, we return $, '', x{:}A'$ which has the needed properties.

□

Lemma [Tagged] 8.6.6 (Properties of VA' algorithm II)
If $\vdash ,$ valid then $VA'(,)$ returns.

Proof: By structural induction on the typing derivation. Example case:

DECL-T: Given $\vdash , ', x{:}A$ valid derived via rule DECL-T from $, ' \vdash A :: \Omega$ and $x \notin \text{dom}(, ')$ $\Rightarrow \vdash , '$ valid (Theorem 6.8.3). Applying the induction hypothesis gives us that $VA'(, ')$ returns some assignment, call it $, ''$. By Lemma 8.6.5, $, ' \rightarrow^* , ''$ and $, ''$ is in canonical form $\Rightarrow \vdash , ''$ valid (Theorem 8.5.6) $\Rightarrow , '' \vdash A :: \Omega$ (Lemma 8.5.5) $\Rightarrow$ $VT'(, '', A)$ always returns (Lemma 8.6.4). Hence, by inspection of VA', we always return.

□

By combining VA' and VC', I can decide constructor validity without any preconditions:

Lemma [Tagged] 8.6.7

1. *If $VA'(,)$ returns then $\vdash VA'(,)$ valid.*
2. *$VT'(VA'(,), A)$ and $VC'(VA'(,), A)$ always terminate.*
3. *$, \vdash A :: \Omega$ iff $VT'(VA'(,), A)$ returns.*
4. *$, \vdash A :: K$ iff $\exists A'.\ VC'(VA'(,), A)$ returns (A', K).*

Proof: Part one follows from Lemma 8.6.5 and Theorem 8.5.6. Part two follows from part one, Lemma 8.6.5, and Lemma 8.6.3. The proofs of parts three and four are similar. We give just the proof of part four here:

$$\begin{array}{rcl}
VA(\,,\,) & = & VA'(\,,\,) \\
VC(\,,\,,A) & = & VC'(VA'(\,,\,),\ A).2 \\
PL(\,,\,,x\rho,A) & = & PL'(VA'(\,,\,),\ x\rho,\ VT'(VA'(\,,\,),A)) \\
EC(\,,\,,A_1,A_2) & = & EC'(VC'(VA'(\,,\,),A_1),\ VC'(VA'(\,,\,),A_2))
\end{array}$$

Figure 8.5: Procedures for deciding judgments

$\Rightarrow$: Given $VC'(VA'(\,,\,),A)$ returns $(A',K) \Rightarrow VA'(\,,\,)$ returns. Hence, by Lemma 8.6.5, $\vdash VA'(\,,\,)$ valid, $\,,\ \rightarrow^* VA'(\,,\,)$, and and $VA'(\,,\,)$ is in canonical form. By part one, $\vdash VA'(\,,\,)$ valid. By Lemma 8.6.3 then, $VA'(\,,\,) \vdash A :: K$. Hence, by Lemma 8.5.5, $\,,\ \vdash A :: K$.

$\Leftarrow$: Given $\,,\ \vdash A :: K \Rightarrow \vdash\,,\,$ valid (Theorem 6.8.3) $\Rightarrow VA'(\,,\,)$ returns (Lemma 8.6.6). Hence, by Lemma 8.6.5, $\vdash VA'(\,,\,)$ valid, $\,,\ \rightarrow^* VA'(\,,\,)$, and and $VA'(\,,\,)$ is in canonical form. By part one, $\vdash VA'(\,,\,)$ valid. Thus, by Lemma 8.5.5, $VA'(\,,\,) \vdash A :: K$. Hence, by Lemma 8.6.4, $\exists A'.\ VC'(VA'(\,,\,),A)$ returns (A',K).

□

Using this technique, I have defined a set of algorithms for deciding each of the judgments without any preconditions in Figure 8.5.

Lemma [Tagged] 8.6.8 (Deciding judgments I)

1. *$VA(\,,\,)$, $VC(\,,\,,A)$, $PL(\,,\,,x\rho,A)$, and $EC(\,,\,,A_1,A_2)$ always terminate.*
2. *$VA(\,,\,)$ returns iff $\vdash\,,\,$ valid.*
3. *$VC(\,,\,,A)$ returns K iff $\,,\ \vdash A :: K$.*
4. *$PL(\,,\,,x\rho,A)$ returns iff $\,,\ \vdash x\rho \Rightarrow A$.*

Proof: By Lemma 8.6.5, $VA'(\,,\,)$ always terminates $\Rightarrow VA(\,,\,)$ always terminates. By Lemmas 8.6.5 and 8.6.6, $VA'(\,,\,)$ returns iff $\vdash\,,\,$ valid $\Rightarrow VA(\,,\,)$ returns iff $\vdash\,,\,$ valid. The termination of $VC(\,,\,,A)$ and the fact that $VC(\,,\,,A)$ returns K iff $\,,\ \vdash A :: K$ follow from inspection of the definition of $VC(\,,\,,A)$ and Lemma 8.6.7. The termination of $PL(\,,\,,x\rho,A)$ and $EC(\,,\,,A_1,A_2)$ follows from the termination of $VA'(\,,\,)$ (Lemma 8.6.5), the termination of $VT'(VA'(\,,\,),A)$ and $VC'(VA'(\,,\,),A)$ (Lemma 8.6.7), and Lemmas 8.6.2 and 8.6.1. The proof of the last part is as follows:

$\Rightarrow$: Given $PL(, , x\rho, A)$ returns $\Rightarrow \exists A'.\ VT'(VA(,), A)$ returns $A' \Rightarrow VA'(,)$ returns. Hence, by Lemma 8.6.3, $VA'(,) \vdash A :: \Omega$, $A \rightarrow^* A'$, and A' is in canonical form. By subject reduction (Corollary 7.6.17), $VA'(,) \vdash A' :: \Omega$. Hence, by Lemma 8.6.2, $VA'(,) \vdash x\rho \Rightarrow A'$. By Theorem 7.8.5, $VA'(,) \vdash A' = A :: \Omega \Rightarrow VA'(,) \vdash x\rho \Rightarrow A$ (P-MOVE). By Lemma 8.6.5, Lemma 8.6.7, and Lemma 8.5.5, $, \vdash x\rho \Rightarrow A$.

$\Leftarrow$: Given $, \vdash x\rho \Rightarrow A \Rightarrow , \vdash A :: \Omega$ (Theorem 7.6.14) $\Rightarrow \exists A'.\ VT'(VA(,), A)$ returns $A' \Rightarrow VA'(,)$ returns. Hence, by Lemma 8.6.3, $VA'(,) \vdash A :: \Omega$, $A \rightarrow^* A'$, and A' is in canonical form. By subject reduction (Corollary 7.6.17), $VA'(,) \vdash A' :: \Omega$. By Lemma 8.6.5, Lemma 8.6.7, and Lemma 8.5.5, $VA'(,) \vdash x\rho \Rightarrow A$. By Theorem 7.8.5, $VA'(,) \vdash A = A' :: \Omega \Rightarrow VA'(,) \vdash x\rho \Rightarrow A'$ (P-MOVE). Hence, by Lemma 8.6.2, $PL'(VA'(,), x\rho, VT'(VA'(,), A))$ returns $\Rightarrow PL(, , x\rho, A)$ returns.

$\Box$

Lemma [Tagged] 8.6.9 (Deciding judgments II)
$EC(, , A_1, A_2)$ returns K iff $, \vdash A_1 = A_2 :: K$.

Proof:

$\Rightarrow$: Given $EC(, , A_1, A_2)$ returns $K \Rightarrow \exists A_1', A_2', K_1, K_2$ such that $VC'(VA'(,), A_1)$ returns (A_1', K_1) and $VC'(VA'(,), A_2)$ returns (A_2', K_2). By Lemma 8.6.7, $, \vdash A_1 :: K_1$ and $, \vdash A_2 :: K_2$. By Lemma 8.6.3, $A_1 \rightarrow^* A_1'$, $A_2 \rightarrow^* A_2'$, and A_1' and A_2' are in canonical form. By subject reduction (Corollary 7.6.17), $, \vdash A_1' :: K_1$ and $, \vdash A_2' :: K_2$. Hence, by Lemma 8.6.1, $, \vdash A_1' = A_2' :: K \Rightarrow , \vdash A_1' :: K$ and $, \vdash A_2' :: K$ (Theorem 6.8.3) $\Rightarrow K = K_1 = K_2$ (Theorem 7.6.11). By Theorem 7.8.5, $, \vdash A_1 = A_1' :: K$ and $, \vdash A_2 = A_2' :: K$. Hence, by E-TRAN and E-SYM, $, \vdash A_1 = A_2 :: K$.

$\Leftarrow$: Given $, \vdash A_1 = A_2 :: K \Rightarrow , \vdash A_1 :: K$ and $, \vdash A_2 :: K$ (Theorem 6.8.3) $\Rightarrow \exists A_1', A_2'$ such that $VC'(VA'(,), A_1)$ returns (A_1', K) and $VC'(VA'(,), A_2)$ returns (A_2', K) (Lemma 8.6.7) By Lemma 8.6.7, Lemma 8.6.5, and Corollary 7.6.17, $, \vdash A_1' :: K$, $, \vdash A_2' :: K$, and A_1' and A_2' are in canonical form. Hence, by Lemma 8.6.1, $EC'(VC'(VA'(,), A_1), VC'(VA'(,), A_2))$ returns $K \Rightarrow EC(, , A_1, A_2)$ returns K.

$\Box$

The consequences of these algorithms can be summed up in three propositions:

Corollary [Tagged] 8.6.10 (Decidability of judgments)
The following judgments are decidable:

1. $\vdash ,$ *valid*

2. $, \vdash A :: K$

3. $, \vdash x\rho \Rightarrow A$

4. $, \vdash A = A' :: K$

Corollary [Tagged] 8.6.11 (Kind Inference)
A recursive algorithm $\mathcal{A}$ *exists such that*

1. *If* $, \vdash A :: K$ *then* $\mathcal{A}(, , A)$ *returns* K.

2. *If* $\neg\exists K. \, , \vdash A :: K$ *then* $\mathcal{A}(, , A)$ *raises fail*

Proof: Let $\mathcal{A}(, , A) = VC(, , A)$. The desired result then follows immediately from Lemma 8.6.8. $\square$

Theorem [Tagged] 8.6.12 (Computability of canonical types)
If $, \vdash A :: \Omega$ *then a constructor* A' *is recursively computable such that* $, \vdash A = A' :: \Omega$ *and* A' *is in canonical form.*

Proof: By Lemma 8.6.7, $VT'(VA'(,), A)$ always returns. By Lemma 8.6.3, $A \rightarrow^* VT'(VA'(,), A)$ and $VT'(VA'(,), A)$ is in canonical form. By Corollary 7.6.17, $, \vdash VT'(VA'(,), A) :: \Omega$. By Theorem 7.8.5, $, \vdash A = VT'(VA'(,), A) :: \Omega$. $\square$

Chapter 9

Types: Summary

In this chapter I transfer the results of the previous two chapters on the tagged system to the original untagged system. I do this by introducing two transforms I call *tag removal* and *stamping*. Tag removal ($\Leftrightarrow^{\ominus}$) transforms objects from the tagged to the untagged system by removing tags (e.g., $x\rho_A\rho'!^{\ominus} = x\rho\rho'!$). Stamping ($\Leftrightarrow^{\oplus,}$) transforms in the opposite direction by adding tags:

$$x\rho!^{\oplus,} = x_{,\,(x)}\rho!$$

The tag to be added is obtained from the current assignment. Using these two transforms, I shall show how to transform judgment derivations between the two systems. Transferring the results between systems is then straightforward.

This chapter is the last of the four chapters devoted to the constructor validity and equality judgments. In the remaining chapters about the kernel system, I shall discuss subtyping, soundness, and type checking terms respectively.

9.1 Tag Removal

In this section I introduce the tag removal transform and prove the needed properties about it. Tag removal can be applied to tagged constructors, declarations, assignments, and judgments, resulting in untagged objects of the same sort. Its definition is straightforward:

Definition [Tagged] 9.1.1 (Tag removal)

$$\begin{array}{rcl}
\alpha^{\ominus} & = & \alpha \\
\mathbf{rec}^{\ominus} & = & \mathbf{rec} \\
\mathbf{ref}^{\ominus} & = & \mathbf{ref} \\
{<}K{>}^{\ominus} & = & {<}K{>} \\
(\Pi x{:}A_1.\, A_2)^{\ominus} & = & \Pi x{:}A_1^{\ominus}.\, A_2^{\ominus} \\
(\Sigma x{:}A_1.\, A_2)^{\ominus} & = & \Sigma x{:}A_1^{\ominus}.\, A_2^{\ominus} \\
(\lambda\alpha{::}K.\, A)^{\ominus} & = & \lambda\alpha{::}K.\, A^{\ominus} \\
(A_1\, A_2)^{\ominus} & = & A_1^{\ominus}\, A_2^{\ominus} \\
{<}{=}A{::}K{>}^{\ominus} & = & {<}{=}A^{\ominus}{::}K{>} \\
x\rho_{A}\rho'!^{\ominus} & = & x\rho\rho'! \\
\\
(\alpha{::}K)^{\ominus} & = & \alpha{::}K \\
(x{:}A)^{\ominus} & = & x{:}A^{\ominus} \\
\\
\bullet^{\ominus} & = & \bullet \\
(,\, , D)^{\ominus} & = & ,\,^{\ominus}, D^{\ominus} \\
\\
(\vdash ,\ \ \mathit{valid})^{\ominus} & = & \vdash ,\,^{\ominus}\ \mathit{valid} \\
(,\ \vdash A :: K)^{\ominus} & = & ,\,^{\ominus} \vdash A^{\ominus} :: K \\
(,\ \vdash x\rho \Rightarrow A)^{\ominus} & = & ,\,^{\ominus} \vdash x\rho \Rightarrow A^{\ominus} \\
(,\ \vdash A_1 = A_2 :: K)^{\ominus} & = & ,\,^{\ominus} \vdash A_1^{\ominus} = A_2^{\ominus} :: K
\end{array}$$

Tag removal interacts with the other operators in the expected ways:

Lemma [Tagged] 9.1.2

1. *$FCV(A^{\ominus}) \subseteq FCV(A)$*
2. *$FTV(A^{\ominus}) \subseteq FTV(A)$*

Lemma [Tagged] 9.1.3 (Properties of tag removal)

1. *$([A_2/\alpha]A_1)^{\ominus} = [A_2^{\ominus}/\alpha](A_1^{\ominus})$*
2. *$([x\rho/x']A)^{\ominus} = [x\rho/x'](A^{\ominus})$*
3. *If $\mathcal{S}(A, x\rho, \rho')$ exists then $\mathcal{S}(A, x\rho, \rho')^{\ominus} = \mathcal{S}(A^{\ominus}, x\rho, \rho')$.*
4. *$dom(,\,) = dom(,\,^{\ominus})$*

5. *If* $D \in ,$ *then* $D^{\ominus} \in ,^{\ominus}$

Proof: Proved sequentially by structural induction on A_1, A, ρ', , , and , respectively. Examples:

$$([A/\alpha]\lambda\alpha'{::}K.\, A')^{\ominus} = (\lambda\alpha'{::}K.\, [A/\alpha]A')^{\ominus} = \lambda\alpha'{::}K.\, ([A/\alpha]A')^{\ominus}$$
$$= \lambda\alpha'{::}K.\, [A^{\ominus}/\alpha](A'^{\ominus}) = [A^{\ominus}/\alpha]\lambda\alpha'{::}K.\, (A'^{\ominus}) = [A^{\ominus}/\alpha]((\lambda\alpha'{::}K.\, A')^{\ominus})$$

$$([A_2/\alpha]x\rho_{A_1}\rho'!)^{\ominus} = (x\rho_{[A_2/\alpha]A_1}\rho'!)^{\ominus} = x\rho\rho'! = [A_2^{\ominus}/\alpha]x\rho\rho'!$$
$$= [A_2^{\ominus}/\alpha](x\rho_{A_1}\rho'!)^{\ominus}$$

$$([x\rho/x']x''\rho'{}_{A}\rho''!)^{\ominus} = (([x\rho/x']x''\rho')_{[x\rho/x']A}\rho'')^{\ominus} = ([x\rho/x']x''\rho')\rho''$$
$$= [x\rho/x']x''\rho'\rho'' = [x\rho/x'](x''\rho'{}_{A}\rho'')^{\ominus}$$

$$\mathcal{S}(\Sigma x'{:}A_1.\, A_2, x\rho, .2\rho')^{\ominus} = \mathcal{S}([x\rho.1/x']A_2, x\rho.2, \rho')^{\ominus} =$$
$$\mathcal{S}(([x\rho.1/x']A_2)^{\ominus}, x\rho.2, \rho') = \mathcal{S}([x\rho.1/x'](A_2^{\ominus}), x\rho.2, \rho') =$$
$$\mathcal{S}(\Sigma x'{:}A_1^{\ominus}.\, A_2^{\ominus}, x\rho, .2\rho') = \mathcal{S}((\Sigma x'{:}A_1.\, A_2)^{\ominus}, x\rho, .2\rho')$$

$$\mathrm{dom}(, , x{:}A) = \mathrm{dom}(,) \cup \mathrm{dom}(x{:}A) = \mathrm{dom}(,) \cup \{x\} = \mathrm{dom}(, ^{\ominus}) \cup \{x\}$$
$$= \mathrm{dom}(, ^{\ominus}) \cup \mathrm{dom}(x{:}A^{\ominus}) = \mathrm{dom}(, ^{\ominus}, (x{:}A)^{\ominus}) = \mathrm{dom}((, , x{:}A)^{\ominus})$$

$$D \in , \Rightarrow , = \bullet, D_1, \ldots, D_m, D, D_{m+1}, \ldots, D_n$$
$$\Rightarrow , ^{\ominus} = \bullet, D_1^{\ominus}, \ldots, D_m^{\ominus}, D^{\ominus}, D_{m+1}^{\ominus}, \ldots, D_n^{\ominus} \Rightarrow D^{\ominus} \in , ^{\ominus}$$

Where $\alpha' \neq \alpha$, $\alpha' \notin \mathrm{FCV}(A) \supseteq \mathrm{FCV}(A^{\ominus})$ by Lemma 9.1.2, and $([x\rho/x']x''\rho')\rho'' = [x\rho/x']x''\rho'\rho''$ by Lemma 6.5.1. □

Lemma [Tagged] 9.1.4 *If* $\mathcal{S}(A^{\ominus}, x\rho, \rho')$ *exists then* $\mathcal{S}(A, x\rho, \rho')$ *exists.*

Proof: By structural induction on ρ'. □

Lemma [Tagged] 9.1.5 $(, ; , ')^{\ominus} = , ^{\ominus}; , '^{\ominus}$

If a tagged constructor extraction $x\rho_{A}\rho'!$ has kind K under , $(, \vdash x\rho_{A}\rho'! :: K)$, then it will always be equal to a constructor extraction involving the same place, but with a tag of , (x) next to x (i.e., $x_{, (x)}\rho\rho'!$). This equality allows the stamping transform to assign tags of this sort to all constructor extractions.

Lemma [Tagged] 9.1.6 *If* $, \vdash x\rho \Rightarrow A$ *and* $, \vdash x\rho\rho' \Rightarrow A'$ *where* A' *has either the form* $<\!K\!>$ *or the form* $<\!=\!A''{::}K\!>$ *for some* A'' *then* $, \vdash x_{, (x)}\rho\rho'! = x\rho_{A}\rho'! :: K$.

Proof: By structural induction on the derivation of $, \vdash x\rho \Rightarrow A$:

P-INIT: Given $, \vdash x \Rightarrow A$ ($\rho = \epsilon$) derived via rule P-INIT from $\vdash ,$ valid and $x{:}A \in ,$. By C-EXT-O2 or C-EXT-T2 (depending on the form of A''), $, \vdash x_{, (x)}\rho\rho'! :: K$. By Lemma 6.8.11, $, (x) = A$. By E-REFL then, $, \vdash x_{, (x)}\rho\rho'! = x\rho_{A}\rho'! :: K$.

P-MOVE: Given $, \vdash x\rho\rho'' \Rightarrow A$ derived via rule P-MOVE from $, \vdash x\rho \Rightarrow A_1$, $, \vdash A_1 = A_2 :: \Omega$, and $\mathcal{S}(A_2, x\rho, \rho'') = A$. Applying the induction hypothesis gives us that $, \vdash x_{, (x)}\rho\rho''\rho'! = x\rho_{A_1}\rho''\rho'! :: K$. By E-EXT2, $, \vdash x\rho_{A_1}\rho''\rho'! = x\rho\rho''_{A}\rho'! :: K$. The desired result then follows by E-TRAN.

□

Applying the equality twice followed by E-TRAN to two different valid tagged constructor extractions involving the same place shows that the two constructor extractions must be equal regardless of what or where their tags are:

Corollary [Tagged] 9.1.7 *If* $, \vdash x\rho_{A_1}\rho'\rho'' :: K$ *and* $, \vdash x\rho\rho'_{A_2}\rho'' :: K$ *then* $, \vdash x\rho_{A_1}\rho'\rho'' = x\rho\rho'_{A_2}\rho'' :: K$.

Proof: Apply Lemma 7.6.5 then Lemma 9.1.6 for each of the two givens then apply E-TRAN and E-SYM. □

This result can be usefully generalized to show that two valid tagged constructors that differ only in their tags' values or locations must be equal:

Theorem [Tagged] 9.1.8 *If* $, \vdash A_1 :: K$, $, \vdash A_2 :: K$, *and* $A_1^{\ominus} = A_2^{\ominus}$ *then* $, \vdash A_1 = A_2 :: K$.

Proof: By structural induction on $A_1^{\ominus}$. Example cases:

Var: Here $A_1^{\ominus} = A_2^{\ominus} = \alpha \Rightarrow A_1 = A_2 = \alpha$ (inspection of the definition of tag removal) $\Rightarrow , \vdash A_1 = A_2 :: K$ (E-REFL).

DFun: Here $A_1^{\ominus} = A_2^{\ominus} = \Pi x{:}B_1.\, B_2$ for some untagged constructors B_1 and B_2, $, \vdash A_1 :: K$, and $, \vdash A_2 :: K \Rightarrow A_1 = \Pi x{:}A_{11}.\, A_{12}$ and $A_2 = \Pi x{:}A_{21}.\, A_{22}$ for some A_{11}, A_{12}, A_{21}, and A_{22} where $A_{11}^{\ominus} = A_{21}^{\ominus} = B_1$ and $A_{12}^{\ominus} = A_{22}^{\ominus} = B_2$ (inspection of the definition of tag removal) $\Rightarrow K = \Omega$, $, \vdash A_{11} :: \Omega$, $, \vdash A_{21} :: \Omega$, $, , x{:}A_{11} \vdash A_{12} :: \Omega$, and $, , x{:}A_{21} \vdash A_{22} :: \Omega$ (C-DFUN). Applying the induction hypothesis gives us that $, \vdash A_{11} = A_{21} :: \Omega \Rightarrow , , x{:}A_{11} \vdash A_{22} :: \Omega$ (Theorem 7.6.9). Applying the induction hypothesis again then gives us that $, , x{:}A_{11} \vdash A_{12} = A_{22} :: \Omega \Rightarrow , \vdash A_1 = A_2 :: K$ (E-DFUN).

Ext: Here $A_1^\ominus = A_2^\ominus = x\rho!$, $, \vdash A_1 :: K$, and $, \vdash A_2 :: K \Rightarrow A_1 = x\rho_1{}_{A_1'}\rho_2\rho_3!$ and $A_2 = x\rho_1\rho_2{}_{A_2'}\rho_3!$ where $\rho_1\rho_2\rho_3 = \rho$ for some A_1' and A_2'
$\Rightarrow , \vdash x\rho_1{}_{A_1'}\rho_2\rho_3! :: K$, and $, \vdash x\rho_1\rho_2{}_{A_2'}\rho_3! :: K \Rightarrow , \vdash A_1 = A_2 :: K$
(Corollary 9.1.7).

$\square$

9.2 Stamping

In this section I introduce the stamping transform and prove the needed properties about it. Stamping can be applied to (untagged) constructors, declarations, assignments, and judgments, resulting in tagged objects of the same sort. The stamping of constructors and declarations takes the stamped version of the current assignment as an extra argument so that the transform can lookup term variables' types for use as tags. The definition of stamping is as follows:

Definition 9.2.1 (Stamping)

$$
\begin{array}{rcll}
\alpha^{\oplus,} & = & \alpha & \\
\textbf{rec}^{\oplus,} & = & \textbf{rec} & \\
\textbf{ref}^{\oplus,} & = & \textbf{ref} & \\
{<}K{>}^{\oplus,} & = & {<}K{>} & \\
(\Pi x{:}A_1.\, A_2)^{\oplus,} & = & \Pi x{:}A_1^{\oplus,}\, .\, A_2^{\oplus,\ ,x:A_1^{\oplus,}} & (x \notin dom(,\ ) \cup FTV(,\ )) \\
(\Sigma x{:}A_1.\, A_2)^{\oplus,} & = & \Sigma x{:}A_1^{\oplus,}\, .\, A_2^{\oplus,\ ,x:A_1^{\oplus,}} & (x \notin dom(,\ ) \cup FTV(,\ )) \\
(\lambda\alpha{::}K.\, A)^{\oplus,} & = & \lambda\alpha{::}K.\, A^{\oplus,\ ,\alpha::K} & (\alpha \notin dom(,\ ) \cup FCV(,\ )) \\
(A_1\, A_2)^{\oplus,} & = & A_1^{\oplus,}\ A_2^{\oplus,} & \\
{<}{=}A{::}K{>}^{\oplus,} & = & {<}{=}A^{\oplus,}\ {::}K{>} & \\
x\rho!^{\oplus,} & = & x_{,\ (x)}\rho! & \\
 & & & \\
(\alpha{::}K)^{\oplus,} & = & \alpha{::}K & \\
(x{:}A)^{\oplus,} & = & x{:}A^{\oplus,} & \\
 & & & \\
\bullet^{\oplus} & = & \bullet & \\
(,\ , D)^{\oplus} & = & ,\ ^{\oplus}, D^{\oplus,\ ^{\oplus}} & \\
 & & & \\
(\vdash ,\ \ valid)^{\oplus} & = & \vdash ,\ ^{\oplus}\ valid & \\
(,\ \vdash A :: K)^{\oplus} & = & ,\ ^{\oplus} \vdash A^{\oplus,\ ^{\oplus}} :: K & \\
(,\ \vdash x\rho \Rightarrow A)^{\oplus} & = & ,\ ^{\oplus} \vdash x\rho \Rightarrow A^{\oplus,\ ^{\oplus}} & \\
(,\ \vdash A_1 = A_2 :: K)^{\oplus} & = & ,\ ^{\oplus} \vdash A_1^{\oplus,\ ^{\oplus}} = A_2^{\oplus,\ ^{\oplus}} :: K &
\end{array}
$$

Stamping is not always defined. In particular, $x\rho!^{\oplus,}$ is only defined if $x \in \mathrm{dom}(,\ )$. Stamping is defined, however, for the components of judgments. This result will be sufficient to translate results between the systems.

Theorem [Mixed] 9.2.2 (Existence of stamping)

1. *If* $FTV(A) \subseteq dom(,')$ *then* $A^{\oplus,\ '}$ *exists.*
2. *If* $,\ ^{\oplus}$ *exists then* $dom(,\ ^{\oplus}) = dom(,\ )$.
3. *If* $\vdash ,\ \ valid$ *then* $,\ ^{\oplus}$ *exists.*

4. *If* $, \vdash A :: K$ *then* $A^{\oplus, ^{\oplus}}$ *exists.*

5. *If* $, \vdash x\rho \Rightarrow A$ *then* $A^{\oplus, ^{\oplus}}$ *exists.*

6. *If* $, \vdash A_1 = A_2 :: K$ *then* $A_i^{\oplus, ^{\oplus}}$ *exist.*

(Here $,'$ *is a tagged assignment; all other variables are untagged.)*

Proof: The first two parts are proved by structural induction on A and $,$ respectively. Then the last four parts are proved by simultaneous structural induction on the typing derivations. In the last three parts, Lemma 6.3.5 plus part three via the induction hypothesis shows that $,^{\oplus}$ exists. Lemma 6.5.6 plus parts one and two then shows that $A^{\oplus, ^{\oplus}}$ or $A_i^{\oplus, ^{\oplus}}$ exists. □

Stamping and tag removal are partial inverses in the sense that stamping followed by tag removal leaves untagged objects unchanged. The reverse operation (tag removal followed by stamping) does change the tagged objects, altering their tags' values and locations. If the original object was a valid tagged constructor then the resulting constructor will at least be equal under the equality relation to the original (Theorem 9.1.8).

Lemma [Mixed] 9.2.3 (Cancellation of stamping)

1. *If* $A^{\oplus, '}$ *exists then* $(A^{\oplus, '})^{\ominus} = A$.

2. *If* $,^{\oplus}$ *exists then* $(,^{\oplus})^{\ominus} = ,$.

3. *If the judgment* $J^{\oplus}$ *exists then* $(J^{\oplus})^{\ominus} = J$.

(Here $,'$ *is a tagged assignment; all other variables are untagged.)*

Proof: Proved sequentially using structural induction on A, $,$, and J respectively. □

The interactions between stamping and the other operators are more complicated than in the tag removal case because of the need to take the assignment into account, the need to worry about when stamping is defined, and the need to keep track of where the tags are. In many cases the operators can be related only by using the equality relation.

For example, when trying to relate place substitution and stamping, we can reason as follows:

$$\begin{array}{lrcl} & [x\rho/x'](x'!^{\oplus, ,x':A';, '}) & ? & ([x\rho/x']x'!)^{\oplus, ;[x\rho/x'], '} \\ \Leftrightarrow & [x\rho/x'](x'_{A'}!) & ? & x\rho!^{\oplus, ;[x\rho/x'], '} \\ \Leftrightarrow & x\rho_{A'}! & ? & x_{(, ;[x\rho/x'], ')(x)}\rho! \end{array}$$

The two sides are clearly not the same because their tags are in different locations. Under appropriate conditions, however, they will be equal under the equality relation (Theorem 9.2.6, below).

Lemma [Mixed] 9.2.4 (Properties of stamping I)

1. *If* $A^{\oplus, {}_1;, {}_3}$ *exists and* $dom(, {}_2) \cap dom(, {}_3) = \emptyset$ *then* $A^{\oplus, {}_1;, {}_2;, {}_3} = A^{\oplus, {}_1;, {}_3}$.
2. *If* $,^{\oplus}$ *exists and* $D \in ,$ *then* $D^{\oplus, ^{\oplus}} \in ,^{\oplus}$
3. *If* $FTV(A_1) \subseteq dom(, {}_1)$, $\alpha \notin FCV(, {}_1)$, *and* $FTV(A_2) \subseteq dom(, {}_1, \alpha{::}K;, {}_2)$ *then* $[A_1^{\oplus, {}_1}/\alpha]A_2^{\oplus, {}_1,\alpha::K;, {}_2} = ([A_1/\alpha]A_2)^{\oplus, {}_1;[A_1^{\oplus, {}_1}/\alpha], {}_2}$.

(Here the $, {}_i$*'s are tagged assignments; all other variables are untagged.)*

Proof: Proved sequentially using structural induction on A, $,$, and A_2 respectively, using Lemmas 7.6.1 and 7.3.9. □

Lemma [Mixed] 9.2.5 *If* $, , x{:}A';, ' \vdash B^{\oplus, ,x:A';, '} :: K$ *and* $, \vdash A' = A'' :: \Omega$ *then* $, , x{:}A';, ' \vdash B^{\oplus, ,x:A';, '} = B^{\oplus, ,x:A'';, '} :: K$.
(Here B's are untagged constructors; all other variables are tagged.)

Proof: By structural induction on B. Example cases:

Var: Since $\alpha^{\oplus, ,x:A';, '} = \alpha = \alpha^{\oplus, ,x:A'';, '}$, applying E-REFL gives the desired result.

Dfun: Let $,'' = , , x{:}A';, '$ and $,''' = , , x{:}A'';, '$.
Given $,'' \vdash \Pi x'{:}B^{\oplus, ''}. B'^{\oplus, '',x':B^{\oplus, ''}} :: \Omega$, $x' \notin$ dom$(, ') \cup$ FTV$(, '') \cup$ FTV$(, ''')$, derived via rule C-DFUN from $,'', x'{:}B^{\oplus, ''} \vdash B'^{\oplus, '',x':B^{\oplus, ''}} :: \Omega$
$\Rightarrow \vdash ,'', x'{:}B^{\oplus, ''}$ valid (Lemma 6.8.3) $\Rightarrow ,'' \vdash B^{\oplus, ''} :: \Omega$ (DECL-T). Applying the induction hypothesis gives us that $,'' \vdash B^{\oplus, ''} = B^{\oplus, '''} :: \Omega$ and
$,'', x'{:}B^{\oplus, ''} \vdash B'^{\oplus, '',x':B^{\oplus, ''}} = B'^{\oplus, ''',x':B^{\oplus, ''}} :: \Omega$. By Theorem 7.6.10 and 7.6.9, $,''', x'{:}B^{\oplus, ''} \vdash B'^{\oplus, ''',x':B^{\oplus, ''}} :: \Omega$ and $,''' \vdash B^{\oplus, ''} = B^{\oplus, '''} :: \Omega$. Hence, applying the induction hypothesis again gives us that
$,''', x'{:}B^{\oplus, ''} \vdash B'^{\oplus, ''',x':B^{\oplus, ''}} = B'^{\oplus, ''',x':B^{\oplus, '''}} :: \Omega$.
Applying Theorem 7.6.10 and 7.6.9 again gives us that
$,'', x'{:}B^{\oplus, ''} \vdash B'^{\oplus, ''',x':B^{\oplus, ''}} = B'^{\oplus, ''',x':B^{\oplus, '''}} :: \Omega$

$\Rightarrow\ ,'', x'{:}B^{\oplus,\ ''} \vdash B'^{\oplus,\ '',x'{:}B^{\oplus,\ ''}} = B'^{\oplus,\ ''',x'{:}B^{\oplus,\ '''}} :: \Omega$ (E-TRAN).
Hence, by E-DFUN,
$,'' \vdash \Pi x'{:}B^{\oplus,\ ''}.\, B'^{\oplus,\ '',x'{:}B^{\oplus,\ ''}} = \Pi x'{:}B^{\oplus,\ '''}.\, B'^{\oplus,\ ''',x'{:}B^{\oplus,\ '''}} :: \Omega$
$\Rightarrow\ ,'' \vdash (\Pi x'{:}B.\, B')^{\oplus,\ ''} = (\Pi x'{:}B.\, B')^{\oplus,\ '''} :: \Omega.$

Ext I: Let $,'' = \ ,\ ,x{:}A';\ ,'$ and $,''' = \ ,\ ,x{:}A'';\ ,'$. Given $,'' \vdash x'_{,\ ''(x')}\rho! :: K$ derived via rule C-EXT-O2 from $,'' \vdash x'\rho \Rightarrow <\!K\!>$ and $,'' \vdash x' \Rightarrow\ ,''(x')$. By weakening (Theorem 6.8.10), $,'' \vdash A' = A'' :: \Omega$. By Lemma 6.8.11 and E-REFL, $\forall x'' \in \mathrm{dom}(,'')$. $,'' \vdash\ ,''(x') =\ ,''(x') :: \Omega$. Hence, $,'' \vdash\ ,''(x') =\ ,'''(x') :: \Omega$. Thus, by E-EXT2, $,'' \vdash x'_{,\ ''(x')}\rho! = x'_{,\ '''(x')}\rho! :: K \Rightarrow\ ,'' \vdash (x'\rho!)^{\oplus,\ ''} = (x'\rho!)^{\oplus,\ '''} :: K.$

□

Theorem [Mixed] 9.2.6 *If* $,\ \vdash x\rho \Rightarrow A'$ *and* $,\ ,x'{:}A';\ ,' \vdash B^{\oplus,\ ,x'{:}A';,\ '} :: K$ *then* $,\ ;[x\rho/x'],\ ' \vdash [x\rho/x'](B^{\oplus,\ ,x'{:}A';,\ '}) = ([x\rho/x']B)^{\oplus,\ ;[x\rho/x'],\ '} :: K.$
(Here B's are untagged constructors; all other variables are tagged.)

Proof: By structural induction on B. Example cases:

Var: By Theorem 7.6.12, $,\ ;[x\rho/x'],\ ' \vdash [x\rho/x']\alpha^{\oplus,\ ,x'{:}A';,\ '} :: K$. Since $\alpha^{\oplus,\ ''} = \alpha$ and $[x\rho/x']\alpha = \alpha$, by E-REFL we have the desired result.

Dfun: Let $,'' = \ ,\ ,x'{:}A';\ ,'$, $\theta = [x\rho/x']$, and $,''' = \ ,\ ;\theta,\ '$. We are given that
$,'' \vdash (\Pi x''{:}B_1.\, B_2)^{\oplus,\ ''} :: K$ where $x'' \notin \mathrm{dom}(,'') \cup \mathrm{FTV}(,'') \cup \mathrm{FTV}(,''') \cup \{x, x'\} \Rightarrow$
$,'' \vdash \Pi x''{:}B_1^{\oplus,\ ''}.\, B_2^{\oplus,\ '',x''{:}B_1^{\oplus,\ ''}} :: K$
$\Rightarrow\ ,'', x''{:}B_1^{\oplus,\ ''} \vdash B_2^{\oplus,\ '',x''{:}B_1^{\oplus,\ ''}} :: K$ (C-DFUN) $\Rightarrow \vdash\ ,'', x''{:}B_1^{\oplus,\ ''}$ valid (Lemma 6.8.3) $\Rightarrow\ ,'' \vdash B_1^{\oplus,\ ''} :: \Omega$ (DECL-T). Applying the induction hypothesis then gives us that
$,''' \vdash \theta B_1^{\oplus,\ ''} = (\theta B_1)^{\oplus,\ '''} :: K$ and
$,''', x''{:}\theta B_1^{\oplus,\ ''} \vdash \theta B_2^{\oplus,\ '',x''{:}B_1^{\oplus,\ ''}} = (\theta B_2)^{\oplus,\ ''',x''{:}\theta B_1^{\oplus,\ ''}} :: K.$
By Lemma 9.2.5,
$,''', x''{:}\theta B_1^{\oplus,\ ''} \vdash (\theta B_2)^{\oplus,\ ''',x''{:}\theta B_1^{\oplus,\ ''}} = (\theta B_2)^{\oplus,\ ''',x''{:}(\theta B_1)^{\oplus,\ '''}} :: K$
$\Rightarrow\ ,''', x''{:}\theta B_1^{\oplus,\ ''} \vdash \theta B_2^{\oplus,\ '',x''{:}B_1^{\oplus,\ ''}} = (\theta B_2)^{\oplus,\ ''',x''{:}(\theta B_1)^{\oplus,\ '''}} :: K$
(E-TRAN). Hence, by E-DFUN,
$,''' \vdash \Pi x''{:}\theta B_1^{\oplus,\ ''}.\, \theta B_2^{\oplus,\ '',x''{:}B_1^{\oplus,\ ''}} = \Pi x''{:}(\theta B_1)^{\oplus,\ '''}.\, (\theta B_2)^{\oplus,\ ''',x''{:}(\theta B_1)^{\oplus,\ '''}} :: K$

$\Rightarrow\ ,''' \vdash \theta\Pi x'':B_1^{\oplus,''}.\, B_2^{\oplus,'',x'':B_1^{\oplus,''}} = (\Pi x'':\theta B_1.\, \theta B_2)^{\oplus,'''} :: K$
$\Rightarrow\ ,''' \vdash \theta(\Pi x'':B_1.\, B_2)^{\oplus,''} = (\theta\Pi x'':B_1.\, B_2)^{\oplus,'''} :: K$

Ext I: Let $,'' = ,\ ,x':A';\ ,'$, $\theta = [x\rho/x']$, and $,''' = ,\ ;\theta,\ '$. We are given that $,'' \vdash (x'\rho'!)^{\oplus,''} :: K$. By Theorem 7.6.12, $,''' \vdash \theta(x'\rho'!^{\oplus,''}) :: K \Rightarrow ,''' \vdash x\rho_{\theta A'}\rho'! :: K$. By Lemmas 7.6.5 and 9.1.6, $,''' \vdash x_{,'''(x)}\rho\rho'! = x\rho_{\theta A'}\rho'! :: K$
$\Rightarrow ,''' \vdash (\theta x'\rho'!)^{\oplus,'''} = \theta(x'\rho'!^{\oplus,''}) :: K$. The desired result follows from E-SYM.

Ext II: Let $,'' = ,\ ,x':A';\ ,'$, $\theta = [x\rho/x']$, and $,''' = ,\ ;\theta,\ '$. We are given that $,'' \vdash (x''\rho'!)^{\oplus,''} :: K$, $x'' \neq x'$. By Theorem 7.6.12, $,''' \vdash \theta(x''\rho'!^{\oplus,''}) :: K$
$\Rightarrow\ ,''' \vdash x''_{\theta,\ ''(x'')}\rho'! :: K$. By repeated use of Lemma 6.3.5, $\vdash ,\ ,x':A'$ valid $\Rightarrow$ $x' \notin \mathrm{dom}(,\ )$ (DECL-T) $\Rightarrow x' \notin \mathrm{FTV}(,\ )$ (Theorem 6.8.5) $\Rightarrow \theta,\ = ,$ (Lemma 7.3.9)
$\Rightarrow \theta,''(x'') = (\theta,'')(x'') = (\theta(,\ ,x':A';\ ,'))(x'') = (\theta,\ ,x':\theta A';\theta,\ ')(x'') = (,\ ;\theta,\ ')(x'')$
$= ,'''(x'')$. Hence, by E-REFL, $,''' \vdash x''_{\theta,\ ''(x'')}\rho'! = x''_{,'''(x'')}\rho'! :: K$
$\Rightarrow ,''' \vdash \theta(x''\rho'!^{\oplus,''}) = (\theta x''\rho'!)^{\oplus,'''} :: K$.

□

Lemma [Mixed] 9.2.7 (Properties of stamping II)
If $, \vdash x\rho \Rightarrow B^{\oplus,}$ *and* $\mathcal{S}(B, x\rho, \rho')^{\oplus,}$ *exists then* $, \vdash x\rho\rho' \Rightarrow \mathcal{S}(B, x\rho, \rho')^{\oplus,}$.
(Here B's are untagged constructors; all other variables are tagged.)

Proof: By structural induction on ρ':

Empty: Since $\rho' = \epsilon$ for this case, the desired result is the first precondition.

Fst: Here $\rho' = .1\rho''$ and $B = \Sigma x':B_1.\, B_2$. By Lemma 6.5.3, $\mathcal{S}(B, x\rho, .1\rho'') = \mathcal{S}(\mathcal{S}(B, x\rho, .1), x\rho.1, \rho'') = \mathcal{S}(B_1, x\rho.1, \rho'')$. Hence, by Lemmas 9.1.4 and 9.2.3, $\mathcal{S}(B^{\oplus,}, x\rho, .1)$ exists. By Theorems 7.6.15 and 7.6.13, $, \vdash x\rho.1 \Rightarrow \mathcal{S}(B^{\oplus,}, x\rho, .1) \Rightarrow , \vdash x\rho.1 \Rightarrow B_1^{\oplus,}$. Applying the induction hypothesis gives us that $, \vdash x\rho\rho.1'' \Rightarrow \mathcal{S}(B_1, x\rho.1, \rho'')^{\oplus,} \Rightarrow , \vdash x\rho\rho.1'' \Rightarrow \mathcal{S}(B, x\rho, \rho.1'')^{\oplus,}$.

Snd: Here $\rho' = .2\rho''$ and $B = \Sigma x':B_1.\, B_2$. By Lemma 6.5.3, $\mathcal{S}(B, x\rho, .2\rho'') = \mathcal{S}(\mathcal{S}(B, x\rho, .2), x\rho.2, \rho'') = \mathcal{S}(([x\rho.1/x']B_2), x\rho.2, \rho'')$. Hence, by Lemmas 9.1.4 and 9.2.3, $\mathcal{S}(B^{\oplus,}, x\rho, .2)$ exists. By Theorem 7.6.15, $, \vdash (\Sigma x':B_1.\, B_2)^{\oplus,} :: \Omega$
$\Rightarrow ,\ ,x':B_1^{\oplus,} \vdash B_2^{\oplus,\ ,x':B_1^{\oplus,}} :: \Omega$ (Lemma 6.8.3 and DECL-T). By Theorem 7.6.13, $, \vdash x\rho.2 \Rightarrow \mathcal{S}(B^{\oplus,}, x\rho, .2)$

$\Rightarrow\ ,\ \vdash x\rho.2 \Rightarrow [x\rho.1/x']B_2^{\oplus,\ ,x':B_1^{\oplus,}}$. By Theorem 9.2.6,
$,\ \vdash [x\rho.1/x']B_2^{\oplus,\ ,x':B_1^{\oplus,}} = ([x\rho.1/x']B_2)^{\oplus,}\ :: \Omega$. Hence, by P-MOVE,
$,\ \vdash x\rho.2 \Rightarrow ([x\rho.1/x']B_2)^{\oplus,}$. Applying the induction hypothesis gives us that
$,\ \vdash x\rho\rho.2'' \Rightarrow \mathcal{S}([x\rho.1/x']B_2, x\rho.2, \rho'')^{\oplus,}\ \Rightarrow\ ,\ \vdash x\rho\rho.2'' \Rightarrow \mathcal{S}(B, x\rho, .2\rho'')^{\oplus,}$.

$\square$

Relating stamping and the append operator for assignments requires defining a notion of *extended* stamping for assignments. Extended stamping $(,\ '^{\oplus,}\ )$ takes an additional argument that represents the result of stamping the preceding assignment. The desired result is then Lemma 9.2.9 below:

Definition [Mixed] 9.2.8 (Extended Stamping)

$$\begin{aligned} \bullet^{\oplus,} &= \bullet \\ (,\ ', D)^{\oplus,} &= ,\ '^{\oplus,}\ , D^{\oplus,\ ;,\ '^{\oplus,}} \end{aligned}$$

(, is a tagged assignment; everything else is from the untagged system.)

Lemma 9.2.9 $(,\ ;\ ,\ ')^{\oplus} = ,\ ^{\oplus};\ ,\ '^{\oplus,\ ,^{\oplus}}$

Lemma [Mixed] 9.2.10 *If* $,\ ^{\oplus,\ ,'}$ *exists then* $dom(,\ ^{\oplus,\ ,'}) = dom(,\ )$.
(Here , is untagged while the ,′ is tagged.)

Lemma [Mixed] 9.2.11 *Suppose* $dom(,_2) \cap (dom(,_3) \cup dom(,\ )) = \emptyset$. *Then if* $,\ ^{\oplus,\ ,_1;\ ,_3}$ *exists,* $,\ ^{\oplus,\ ,_1;\ ,_2;\ ,_3} = ,\ ^{\oplus,\ ,_1;\ ,_3}$.
(Here the , is a tagged assignment; all other variables are untagged.)

Proof: By structural induction on , using Lemmas 9.2.4, 6.8.9, and 6.8.2 for the base case. $\square$

9.3 System Correspondence

In this section I show that tag removal can be used to transform tagged judgment derivations into derivations of the corresponding untagged judgments and that stamping can be used to transform untagged judgment derivations into derivations of the corresponding tagged judgments. These results mean that the original judgment J is true iff the tagged judgment $J^{\oplus}$ exists and is true.

Theorem [Tagged] 9.3.1 (System correspondence I)

1. *If* $\vdash ,$ *valid then* $(\vdash ,$ *valid*$)^\ominus$.
2. *If* $, \vdash A :: K$ *then* $(, \vdash A :: K)^\ominus$.
3. *If* $, \vdash x\rho \Rightarrow A$ *then* $(, \vdash x\rho \Rightarrow A)^\ominus$.
4. *If* $, \vdash A_1 = A_2 :: K$ *then* $(, \vdash A_1 = A_2 :: K)^\ominus$.

Proof: By simultaneous structural induction on the length of the typing derivations, using lemma 9.1.3 as needed. Example cases:

DECL-T: Given $\vdash , , x{:}A$ valid derived via rule DECL-T from $, \vdash A :: \Omega$ and $x \notin \mathrm{dom}(,)$. Applying the induction hypothesis gives us that $,^\ominus \vdash A^\ominus :: \Omega$. By Lemma 9.1.3, $x \notin \mathrm{dom}(,^\ominus)$ so by rule DECL-T, we have that $\vdash ,^\ominus, x{:}A^\ominus$ valid $\Rightarrow (\vdash , , x{:}A$ valid$)^\ominus$.

C-EXT-T2: Given $, \vdash x\rho_{A}\rho'! :: K$ derived via rule C-EXT-T2 from $, \vdash x\rho\rho' \Rightarrow \texttt{<=}A'{::}K\texttt{>}$ and $, \vdash x\rho \Rightarrow A$. Applying the induction hypothesis gives us that $,^\ominus \vdash x\rho\rho' \Rightarrow \texttt{<=}A'^\ominus{::}K\texttt{>}$. Hence, by C-EXT-T, $,^\ominus \vdash x\rho\rho'! :: K \Rightarrow (, \vdash x\rho_{A}\rho'! :: K)^\ominus$.

P-INIT: Given $, \vdash x \Rightarrow A$ derived via rule P-INIT from $\vdash ,$ valid and $x{:}A \in ,$. Applying the induction hypothesis gives us that $\vdash ,^\ominus$ valid. By Lemma 9.1.3, $x{:}A^\ominus \in ,^\ominus$. Hence, by P-INIT, $,^\ominus \vdash x \Rightarrow A^\ominus \Rightarrow (, \vdash x \Rightarrow A)^\ominus$.

E-BETA: Given $, \vdash (\lambda\alpha{::}K.\, A)\, A' = [A'/\alpha]A :: K'$ derived via rule E-BETA from $, , \alpha{::}K \vdash A :: K'$ and $, \vdash A' :: K$. Applying the induction hypothesis gives us that $,^\ominus, \alpha{::}K \vdash A^\ominus :: K'$ and $,^\ominus \vdash A'^\ominus :: K$. Since $[A'^\ominus/\alpha]A^\ominus = ([A'/\alpha]A)^\ominus$ by Lemma 9.1.3, by rule E-BETA we have that $,^\ominus \vdash ((\lambda\alpha{::}K.\, A)\, A')^\ominus = ([A'/\alpha]A)^\ominus :: K'$ $\Rightarrow (, \vdash (\lambda\alpha{::}K.\, A)\, A' = [A'/\alpha]A :: K')^\ominus$.

E-EXT2: Given $, \vdash x\rho_{A_1}\rho'\rho''! = x\rho\rho'_{A_2}\rho''! :: K$ derived via rule E-EXT2 from $, \vdash x\rho_{A_1}\rho'\rho''! :: K$, $, \vdash A_1 = A :: \Omega$, and $\mathcal{S}(A, x\rho, \rho') = A_2$. Applying the induction hypothesis gives us that $,^\ominus \vdash x\rho\rho'\rho''! :: K$. Hence, by E-REFL, $,^\ominus \vdash x\rho\rho'\rho''! = x\rho\rho'\rho''! :: K \Rightarrow (, \vdash x\rho_{A_1}\rho'\rho''! = x\rho\rho'_{A_2}\rho''! :: K)^\ominus$.

E-ABBREV2: Given $, \vdash x\rho_{A}! = A' :: K$ derived via rule E-ABBREV2 from $, \vdash x\rho_{A}! :: K$ and $, \vdash A = \texttt{<=}A'{::}K'\texttt{>} :: \Omega$. By Lemma 7.6.5, $, \vdash x\rho \Rightarrow A$ is a sub-derivation of $, \vdash x\rho_{A}! :: K$. Applying the induction hypothesis then gives us that $,^\ominus \vdash x\rho! :: K$, $,^\ominus \vdash x\rho \Rightarrow A^\ominus$, and $,^\ominus \vdash A^\ominus = \texttt{<=}A'^\ominus{::}K'\texttt{>} :: \Omega$. Hence, by E-ABBREV, $,^\ominus \vdash x\rho! = A'^\ominus :: K \Rightarrow (, \vdash x\rho_{A}! = A' :: K)^\ominus$.

□

Theorem 9.3.2 (System correspondence II)

1. *If* $\vdash ,$ *valid then* $(\vdash ,$ *valid*$)^{\oplus}$.
2. *If* $, \vdash A :: K$ *then* $(, \vdash A :: K)^{\oplus}$.
3. *If* $, \vdash x\rho \Rightarrow A$ *then* $(, \vdash x\rho \Rightarrow A)^{\oplus}$.
4. *If* $, \vdash A_1 = A_2 :: K$ *then* $(, \vdash A_1 = A_2 :: K)^{\oplus}$.

Proof: By simultaneous structural induction on the derivations. Existence of the various stamped items is shown using Theorem 9.2.2. Example cases:

DECL-T: Given $\vdash , , x{:}A$ valid derived via rule DECL-T from $, \vdash A :: \Omega$ and $x \notin \mathrm{dom}(,)$. Applying the induction hypothesis gives us that $,^{\oplus} \vdash A^{\oplus,^{\oplus}} :: \Omega$. By Theorem 9.2.2, $\mathrm{dom}(,) = \mathrm{dom}(,^{\oplus}) \Rightarrow x \notin \mathrm{dom}(,^{\oplus})$. Hence, by DECL-T, $\vdash ,^{\oplus}, x{:}A^{\oplus,^{\oplus}}$ valid $\Rightarrow$ $\vdash (, , x{:}A)^{\oplus}$ valid.

C-DFUN: Given $, \vdash \Pi x{:}A.\, A' :: \Omega$ derived via rule C-DFUN from $, , x{:}A \vdash A' :: \Omega$. Applying the induction hypothesis gives us that $,^{\oplus}, x{:}A^{\oplus,^{\oplus}} \vdash A'^{\oplus,^{\oplus}, x:A^{\oplus,^{\oplus}}} :: \Omega$
$\Rightarrow ,^{\oplus} \vdash \Pi x{:}A^{\oplus,^{\oplus}}.\, A'^{\oplus,^{\oplus}, x:A^{\oplus,^{\oplus}}} :: \Omega$ (C-DFUN) $\Rightarrow ,^{\oplus} \vdash (\Pi x{:}A.\, A')^{\oplus,^{\oplus}} :: \Omega$.

C-EXT-T: Given $, \vdash x\rho! :: K$ derived from $, \vdash x\rho \Rightarrow \, <\!=\!A{::}K\!>$. Applying the induction hypothesis gives us that $,^{\oplus} \vdash x\rho \Rightarrow \, <\!=\!A^{\oplus,^{\oplus}}{::}K\!>$. By Theorem 6.8.3, $\vdash ,^{\oplus}$ valid. By Theorem 6.8.5, $x \in \mathrm{dom}(,^{\oplus})$. Hence, by P-INIT, $,^{\oplus} \vdash x \Rightarrow ,^{\oplus}(x)$. By C-EXT-T2 then, $,^{\oplus} \vdash x_{,^{\oplus}(x)}\rho! :: K \Rightarrow ,^{\oplus} \vdash (x\rho!)^{\oplus,^{\oplus}} :: K$.

P-INIT: Given $, \vdash x \Rightarrow A$ derived via rule P-INIT from $\vdash ,$ valid and $x{:}A \in ,$. Applying the induction hypothesis gives us that $\vdash ,^{\oplus}$ valid. By Lemma 9.2.4, $x{:}A^{\oplus,^{\oplus}} \in ,^{\oplus}$. Hence, by P-INIT, $,^{\oplus} \vdash x \Rightarrow A^{\oplus,^{\oplus}}$.

P-MOVE: Given $, \vdash x\rho\rho' \Rightarrow A''$ derived via rule P-MOVE from $, \vdash x\rho \Rightarrow A$, $, \vdash A = A' :: \Omega$, and $\mathcal{S}(A', x\rho, \rho') = A''$. Applying the induction hypothesis gives us that $,^{\oplus} \vdash x\rho \Rightarrow A^{\oplus,^{\oplus}}$ and $,^{\oplus} \vdash A^{\oplus,^{\oplus}} = A'^{\oplus,^{\oplus}} :: \Omega \Rightarrow ,^{\oplus} \vdash x\rho \Rightarrow A'^{\oplus,^{\oplus}}$ (P-MOVE). By Theorem 9.2.2, $A''^{\oplus,^{\oplus}} = \mathcal{S}(A', x\rho, \rho')^{\oplus,^{\oplus}}$ exists. Hence, by Theorem 9.2.7, $,^{\oplus} \vdash x\rho\rho' \Rightarrow A''^{\oplus,^{\oplus}}$.

E-BETA: Given $, \vdash (\lambda\alpha{::}K.\,A)\,A' = [A'/\alpha]A :: K'$ derived via rule E-BETA from $, \alpha{::}K \vdash A :: K'$ and $, \vdash A' :: K \Rightarrow \mathrm{FTV}(A) \subseteq \mathrm{dom}(, \alpha{::}K)$ and $\mathrm{FTV}(A') \subseteq \mathrm{dom}(\,)$ (Theorem 6.5.6). By Theorems 6.3.5 and 6.5.9, $\alpha \notin \mathrm{FCV}(\,)$. Hence, by Theorem 9.2.4, $[A'^{\oplus,}\,/\alpha]A^{\oplus,\;,\alpha::K} = ([A'/\alpha]A)^{\oplus,}$. Applying the induction hypothesis gives us that $,^{\oplus}, \alpha{::}K \vdash A^{\oplus,\;{}^{\oplus},\alpha::K} :: K'$ and $,^{\oplus} \vdash A'^{\oplus,\;{}^{\oplus}} :: K$ $\Rightarrow\; ,^{\oplus} \vdash (\lambda\alpha{::}K.\,A^{\oplus,\;{}^{\oplus},\alpha::K})A'^{\oplus,\;{}^{\oplus}} = [A'^{\oplus,\;{}^{\oplus}}/\alpha](A^{\oplus,\;{}^{\oplus},\alpha::K}) :: K'$ (E-BETA) $\Rightarrow\; ,^{\oplus} \vdash ((\lambda\alpha{::}K.\,A)\,A')^{\oplus,\;{}^{\oplus}} = ([A'/\alpha]A)^{\oplus,\;{}^{\oplus}} :: K'$.

E-ETA: Given $, \vdash \lambda\alpha{::}K.\,A\,\alpha = A :: K{\Rightarrow}K'$, $\alpha \notin \mathrm{dom}(\,)$, derived via rule E-ETA from $, \vdash A :: K{\Rightarrow}K'$ and $\alpha \notin \mathrm{FCV}(A)$. Applying the induction hypothesis gives us that $,^{\oplus} \vdash A^{\oplus,\;{}^{\oplus}} :: K{\Rightarrow}K'$. By Theorem 9.2.2, $\mathrm{dom}(\,) = \mathrm{dom}(,^{\oplus}) \Rightarrow \alpha \notin \mathrm{dom}(,^{\oplus})$. By Theorem 6.8.7, $\mathrm{FCV}(A^{\oplus,\;{}^{\oplus}}) \subseteq \mathrm{dom}(,^{\oplus}) \Rightarrow \alpha \notin \mathrm{FCV}(A^{\oplus,\;{}^{\oplus}})$. By Lemma 9.2.4, $A^{\oplus,\;{}^{\oplus}} = A^{\oplus,\;{}^{\oplus},\alpha::K}$. Hence, by E-ETA, $,^{\oplus} \vdash \lambda\alpha{::}K.\,A^{\oplus,\;{}^{\oplus},\alpha::K}\,\alpha = A^{\oplus,\;{}^{\oplus}} :: K{\Rightarrow}K'$ $\Rightarrow\; ,^{\oplus} \vdash (\lambda\alpha{::}K.\,A\,\alpha)^{\oplus,\;{}^{\oplus}} = A^{\oplus,\;{}^{\oplus}} :: K{\Rightarrow}K'$

E-ABBREV: Given $, \vdash x\rho! = A' :: K$ derived via rule E-ABBREV from $, \vdash x\rho! :: K$, $, \vdash x\rho \Rightarrow A$, and $, \vdash A = \,<\!=\!A'{::}K'\!>\, :: \Omega$. Applying the induction hypothesis gives us that $,^{\oplus} \vdash x_{,\,^{\oplus}(x)}\rho! :: K$, $,^{\oplus} \vdash x\rho \Rightarrow A^{\oplus,\;{}^{\oplus}}$, and $,^{\oplus} \vdash A^{\oplus,\;{}^{\oplus}} = \,<\!=\!A'^{\oplus,\;{}^{\oplus}}{::}K'\!>\, :: \Omega$. By Lemmas 7.6.5 and 9.1.6, $,^{\oplus} \vdash x_{,\,^{\oplus}(x)}\rho! = x\rho_{A^{\oplus,}\;^{\oplus}}! :: K$. By Theorem 7.6.10 and E-ABBREV2, $,^{\oplus} \vdash x\rho_{A^{\oplus,}\;^{\oplus}}! = A'^{\oplus,\;{}^{\oplus}} :: K$. The desired result then follows via E-TRAN.

□

Theorem 9.3.3 (System correspondence III)

1. $\vdash \,,$ *valid iff* $\vdash \,,^{\oplus}$ *valid.*

2. $, \vdash A :: K$ *iff* $,^{\oplus} \vdash A^{\oplus,\;{}^{\oplus}} :: K$.

3. $, \vdash x\rho \Rightarrow A$ *iff* $,^{\oplus} \vdash x\rho \Rightarrow A^{\oplus,\;{}^{\oplus}}$.

4. $, \vdash A_1 = A_2 :: K$ *iff* $,^{\oplus} \vdash A_1^{\oplus,\;{}^{\oplus}} = A_2^{\oplus,\;{}^{\oplus}} :: K$.

Proof: Follows from the two previous system correspondence theorems and the cancellation of stamping lemma. □

9.4 Converted Results

In this section I use the results of the previous three sections to transfer many results for the tagged system, which were proved in the previous three chapters, to the original untagged system. The following results come from Chapter 6:

Theorem 9.4.1 (Weakening)
Suppose $\vdash\ ,\ ;\ ,'$ *valid and* $dom(\ ,') \cap dom(\ ,'') = \emptyset$*. Then:*

1. *If* $\vdash\ ,\ ;\ ,''$ *valid then* $\vdash\ ,\ ;\ ,';\ ,''$ *valid.*

2. *If* $,\ ;\ ,'' \vdash A :: K$ *then* $,\ ;\ ,';\ ,'' \vdash A :: K$.

3. *If* $,\ ;\ ,'' \vdash x\rho \Rightarrow A$ *then* $,\ ;\ ,';\ ,'' \vdash x\rho \Rightarrow A$.

4. *If* $,\ ;\ ,'' \vdash A_1 = A_2 :: K$ *then* $,\ ;\ ,';\ ,'' \vdash A_1 = A_2 :: K$.

Proof: The proof of each part is similar. I give here only the proof for part one. Given $\vdash\ ,\ ;\ ,'$ valid and $\vdash\ ,\ ;\ ,''$ valid $\Rightarrow \vdash (\ ,\ ;\ ,')^{\oplus}$ valid and $\vdash (\ ,\ ;\ ,'')^{\oplus}$ valid (Theorem 9.3.3) $\Rightarrow \vdash\ ,^{\oplus};\ ,'^{\oplus,\ ^{\oplus}}$ valid and $\vdash\ ,^{\oplus};\ ,''^{\oplus,\ ^{\oplus}}$ valid (Lemma 9.2.9) $\Rightarrow$ dom$(\ ,'^{\oplus,\ ^{\oplus}}) =$ dom$(\ ,')$ and dom$(\ ,''^{\oplus,\ ^{\oplus}}) =$ dom$(\ ,'')$ (Lemma 9.2.10) $\Rightarrow$ dom$(\ ,'^{\oplus,\ ^{\oplus}}) \cap$ dom$(\ ,''^{\oplus,\ ^{\oplus}}) = \emptyset \Rightarrow$ $\vdash\ ,^{\oplus};\ ,'^{\oplus,\ ^{\oplus}};\ ,''^{\oplus,\ ^{\oplus}}$ valid (Theorem 6.8.10) $\Rightarrow \vdash (\ ,^{\oplus};\ ,'^{\oplus,\ ^{\oplus}};\ ,''^{\oplus,\ ^{\oplus}})^{\ominus}$ valid (Theorem 9.3.1)
$\Rightarrow \vdash (\ ,^{\oplus})^{\ominus};(\ ,'^{\oplus,\ ^{\oplus}})^{\ominus};(\ ,''^{\oplus,\ ^{\oplus}})^{\ominus}$ valid (Lemma 9.1.5)
$\Rightarrow \vdash\ ,\ ;\ ,';\ ,''$ valid (Lemma 9.2.3) □

Lemma 9.4.2 (Valid assignment properties)
Suppose $\vdash\ ,$ *valid. Then:*

1. *If* $\alpha{::}K \in\ ,$ *and* $\alpha{::}K' \in\ ,$ *then* $K = K'$.

2. *If* $x{:}A \in\ ,$ *then* $,(x) = A$.

3. *If* $x \in dom(\ ,)$ *then* $,\ \vdash\ ,(x) :: \Omega$.

Proof: Inspection of the typing rules for assignments combined with the results from Lemma 6.8.3 reveal that valid assignments never redeclare variables. The last part requires weakening (Theorem 9.4.1) plus repeated use of Lemma 6.3.5. □

The next set of results are from Chapter 7:

Theorem 9.4.3 (Validity of equal constructors)
If $,\ \vdash A_1 = A_2 :: K$ *then* $,\ \vdash A_1 :: K$ *and* $,\ \vdash A_2 :: K$.

Theorem 9.4.4 (Validity of looked up places)
If $, \vdash x\rho \Rightarrow A$ *then* $, \vdash A :: \Omega$.

Theorem 9.4.5 (Uniqueness of constructor kind)
If $, \vdash A :: K_1$ *and* $, \vdash A :: K_2$ *then* $K_1 = K_2$.

Proof: By Theorems 7.6.11 and 9.3.3. □

Theorem 9.4.6 (Replacement by an equal type)
Suppose $, \vdash A_1 = A_2 :: \Omega$. *Then:*

1. *If* $\vdash , , x{:}A_1; ,'$ *valid then* $\vdash , , x{:}A_2; ,'$ *valid.*

2. *If* $, , x{:}A_1; ,' \vdash A :: K$ *then* $, , x{:}A_2; ,' \vdash A :: K$.

3. *If* $, , x{:}A_1; ,' \vdash x'\rho \Rightarrow A$ *then* $, , x{:}A_2; ,' \vdash x'\rho \Rightarrow A$.

4. *If* $, , x{:}A_1; ,' \vdash A = A' :: K$ *then* $, , x{:}A_2; ,' \vdash A = A' :: K$.

Proof: By Theorem 9.4.3, $, \vdash A_2 :: \Omega$. Convert to the tagged system via stamping (Theorem 9.3.3 and Lemma 9.2.9), then apply the tagged version of the theorem (Theorem 7.6.9), then return to the original system via tag removal and cancellation (Theorem 9.3.1, Lemma 9.2.3, Lemma 9.1.3, and Lemma 9.1.5). □

Theorem 9.4.7 (Validity of place substitution)
Suppose $, \vdash x\rho \Rightarrow A'$. *Then:*

1. *If* $\vdash , , x'{:}A'; ,'$ *valid then* $\vdash , ; [x\rho/x'] ,'$ *valid.*

2. *If* $, , x'{:}A'; ,' \vdash A :: K$ *then* $, ; [x\rho/x'] ,' \vdash [x\rho/x']A :: K$.

3. *If* $, , x'{:}A'; ,' \vdash x''\rho'' \Rightarrow A$ *then* $, ; [x\rho/x'] ,' \vdash [x\rho/x']x''\rho'' \Rightarrow [x\rho/x']A$.

4. *If* $, , x'{:}A'; ,' \vdash A_1 = A_2 :: K$ *then* $, ; [x\rho/x'] ,' \vdash [x\rho/x']A_1 = [x\rho/x']A_2 :: K$.

Proof: Convert to the tagged system via stamping (Theorem 9.3.3 and Lemma 9.2.9), then apply the tagged version of the theorem (Theorem 7.6.12), then return to the original system via tag removal and cancellation (Theorem 9.3.1, Lemma 9.2.3, Lemma 9.1.3, and Lemma 9.1.5). □

Theorem 9.4.8 (Strengthening)
If $, _1, x{:}A; , _2 \vdash A_1 = A_2 :: K$*,* $, _1; , _2 \vdash A_1 :: K$*, and* $, _1; , _2 \vdash A_2 :: K$ *then* $, _1; , _2 \vdash A_1 = A_2 :: K$*.*

Proof: Convert to the tagged system via stamping (Theorem 9.3.3 and Lemma 9.2.9), observe that $,_2^{\oplus(, _1, x:A)^\oplus} = ,_2^\oplus, _1^\oplus$ and $A_i^{\oplus(, _1, x:A; , _2)^\oplus} = A_i^{\oplus(, _1; , _2)^\oplus}$ (Lemmas 9.2.10, 6.8.9, 6.8.3, 9.2.11, and 9.2.4), then apply the tagged version of the lemma (Lemma 7.8.6), then return to the original system via tag removal and cancellation (Theorem 9.3.1 and Lemma 9.2.3). $\Box$

Lemma 9.4.9 (Equality of forms)
Suppose $, \vdash A_1 = A_2 :: K$ *and* A_2 *has neither the form* $A_2' \, A_2''$*,* $A_2' \notin \{\mathbf{ref}, \mathbf{rec}\}$*, the form* $x\rho!$*, nor the form* $\lambda\alpha{::}K.\, A_1'$*. Then*

1. *If* A_1 *has one of the forms* α*,* $<K>$*,* $\mathbf{rec}$*, or* $\mathbf{ref}$ *then* $A_1 = A_2$*.*
2. *If* A_1 *has the form* $\Pi x{:}A_1'.\, A_1''$ *then* A_2 *has the form* $\Pi x{:}A_2'.\, A_2''$*.*
3. *If* A_1 *has the form* $\Sigma x{:}A_1'.\, A_1''$ *then* A_2 *has the form* $\Sigma x{:}A_2'.\, A_2''$*.*
4. *If* A_1 *has the form* $<=A_1'{::}K>$ *then* A_2 *has the form* $<=A_2'{::}K>$*.*
5. *If* A_1 *has the form* $\mathbf{ref}\, A_1'$ *then* A_2 *has the form* $\mathbf{ref}\, A_2'$*.*
6. *If* A_1 *has the form* $\mathbf{rec}\, A_1'$ *then* A_2 *has the form* $\mathbf{rec}\, A_2'$*.*

Proof: Convert to the tagged system via stamping (Theorem 9.3.3), then apply the tagged version of the lemma (Lemma 7.8.7), then return to the original system via tag removal and cancellation (Theorem 9.3.1 and Lemma 9.2.3). Note that both stamping and tag removal preserve forms. $\Box$

Lemma 9.4.10 (Component-wise equality)

1. *If* $, \vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega$*,* $x \notin dom(,)$*, then* $, \vdash A_1 = A_1' :: \Omega$ *and* $, , x{:}A_1 \vdash A_2 = A_2' :: \Omega$*.*
2. *If* $, \vdash \Sigma x{:}A_1.\, A_2 = \Sigma x{:}A_1'.\, A_2' :: \Omega$*,* $x \notin dom(,)$*, then* $, \vdash A_1 = A_1' :: \Omega$ *and* $, , x{:}A_1 \vdash A_2 = A_2' :: \Omega$*.*
3. *If* $, \vdash <K> = <K'> :: \Omega$ *then* $K = K'$*.*

4. *If* $, \vdash \mathord{<}{=}A{::}K\mathord{>} = \mathord{<}{=}A'{::}K'\mathord{>} :: \Omega$ *then*
   $K = K'$ *and* $, \vdash A = A' :: K$.

5. *If* $, \vdash \mathbf{ref}\, A = \mathbf{ref}\, A' :: \Omega$ *then* $, \vdash A = A' :: \Omega$.

6. *If* $, \vdash \mathbf{rec}\, A = \mathbf{rec}\, A' :: \Omega$ *then* $, \vdash A = A' :: \Omega{\Rightarrow}\Omega$.

Proof: Convert to the tagged system via stamping (Theorems 9.3.3 and 9.2.2), then apply the tagged version of the lemma (Lemma 7.8.8), then return to the original system via tag removal and cancellation (Theorem 9.3.1 and Lemma 9.2.3). □

Finally, these results are from Chapter 8:

Theorem 9.4.11 (Decidability of judgments)
The following judgments are decidable:

1. $\vdash\, ,$ *valid*

2. $, \vdash A :: K$

3. $, \vdash x\rho \Rightarrow A$

4. $, \vdash A = A' :: K$

Proof: Follows from Theorem 9.3.3, Corollary 8.6.10, and the fact that the stamping process always terminates. The later is proved by structural induction on the item being stamped. □

Corollary 9.4.12 (Kind Inference) *A recursive algorithm $\mathcal{A}$ exists such that*

1. *If* $, \vdash A :: K$ *then* $\mathcal{A}(, , A)$ *returns* K.

2. *If* $\neg\exists K.\ , \vdash A :: K$ *then* $\mathcal{A}(, , A)$ *raises fail*

Proof: Follows from Theorem 9.3.3, Corollary 8.6.11, and the fact that the stamping process always terminates. The later is proved by structural induction on the item being stamped. □

9.5 Semi-Canonical Types

In this section I transfer the useful results about tagged constructors in canonical form from Chapter 8 to the original untagged system. I shall do this by introducing *semi-canonical types*, an untagged analogue of types in canonical form. A type is a semi-canonical type (roughly) if it is the result of removing the tags from a valid tagged type in canonical form:

Definition 9.5.1 (Semi-canonical Types)
A is a semi-canonical type under assignment $,$ $(A \in C_{,})$ iff there exists a tagged constructor B such that:

1. $B^{\ominus} = A$
2. $,^{\oplus} \vdash B :: \Omega$
3. *B is in canonical form*

Note that unlike in the tagged case, the notion of a semi-canonical type is relative to an assignment. For example, $x!$ is a semi-canonical type under $x{:}{<}\Omega{>}$ but not under $x{:}{<}{=}{<}\Omega{>}{::}\Omega{>}$. This reflects the fact that equality (and hence any related notion of rewriting) depends on the assignment in the untagged system but not in the tagged system.

Some properties about semi-canonical types follow immediately from the definition. Note that stamping a semi-canonical type does not necessarily yield a canonical type because stamping does not in general attach the right tags.

Lemma 9.5.2 (Semi-canonical properties)
If $A \in C_{,}$ then

1. $, \vdash A :: \Omega$
2. $,^{\oplus} \vdash B = A^{\oplus ,^{\oplus}} :: \Omega$

where B is the tagged constructor from the definition of $A \in C_{,}$.

Proof: By the definition of $A \in C_{,}$, $B^{\ominus} = A$ and $,^{\oplus} \vdash B :: \Omega$. By Theorem 9.3.1 and Lemma 9.2.3, $, \vdash B^{\ominus} :: \Omega \Rightarrow , \vdash A :: \Omega \Rightarrow ,^{\oplus} \vdash A^{\oplus ,^{\oplus}} :: \Omega$ (Theorem 9.3.3). Hence, by Theorem 9.1.8 and Lemma 9.2.3, $,^{\oplus} \vdash B = A^{\oplus ,^{\oplus}} :: \Omega$. □

In order to keep the proofs modular, I shall treat the definition of semi-canonical types like an abstract data type; that is, I shall provide sufficient propositions about semi-canonical types in this section so that it will not be necessary to refer to the tagged system or the details of this definition again after this chapter.

The following two lemmas formalize the fact that semi-canonical sums are built from semi-canonical components. The equivalent statements for the tagged system involving canonical forms follow immediately from the definition of canonical form.

Lemma 9.5.3 *If $\Sigma x{:}A_1.\, A_2 \in C_\Gamma$, $x \notin dom(\Gamma)$ then $A_1 \in C_\Gamma$ and $A_2 \in C_{\Gamma,x:A_1}$.*

Proof: By the definition of $\Sigma x{:}A_1.\, A_2 \in C_\Gamma$, there exists a tagged constructor B such that $B^\ominus = \Sigma x{:}A_1.\, A_2$, $\Gamma^\oplus \vdash B :: \Omega$, and B is in canonical form $\Rightarrow B = \Sigma x{:}B_1.\, B_2$ for some tagged constructors B_1 and B_2 such that $B_1^\ominus = A_1$ and $B_2^\ominus = A_2$ (inspection of the definition of tag removal) $\Rightarrow \Gamma^\oplus \vdash B_1 :: \Omega$ and $\Gamma^\oplus, x{:}B_1 \vdash B_2 :: \Omega$ (C-DSUM and Theorem 9.2.2) By inspection of the definition of canonical form, B_1 and B_2 must also in canonical form because they are sub-constructors of a canonical constructor $\Rightarrow A_1 \in C_\Gamma$. By Lemma 9.5.2, $\Gamma \vdash \Sigma x{:}A_1.\, A_2 :: \Omega \Rightarrow \Gamma \vdash A_1 :: \Omega$ (C-DSUM) $\Rightarrow \Gamma^\oplus \vdash A_1^{\oplus,\Gamma^\oplus} :: \Omega$ (Theorem 9.3.3) $\Rightarrow \Gamma^\oplus \vdash A_1^{\oplus,\Gamma^\oplus} = B_1 :: \Omega$ (Theorem 9.1.8 and Lemma 9.2.3) $\Rightarrow \Gamma^\oplus, x{:}A_1^{\oplus,\Gamma^\oplus} \vdash B_2 :: \Omega$ (Theorem 9.4.6) $\Rightarrow (\Gamma, x{:}A_1)^\oplus \vdash B_2 :: \Omega \Rightarrow A_2 \in C_{\Gamma,x:A_1}$. □

Lemma 9.5.4 *If $A_1 \in C_\Gamma$, $A_2 \in C_{\Gamma,x:A_1}$, $x \notin dom(\Gamma)$, then $\Sigma x{:}A_1.\, A_2 \in C_\Gamma$.*

Proof: By the definition of semi-canonical types, $\exists$ tagged constructors B_1 and B_2 such that $B_1^\ominus = A_1$, $B_2^\ominus = A_2$, $\Gamma^\oplus \vdash B_1 :: \Omega$, $\Gamma^\oplus, x{:}A_1^{\oplus,\Gamma^\oplus} \vdash B_2 :: \Omega$, and B_1 and B_2 are in canonical form $\Rightarrow \Sigma x{:}B_1.\, B_2$ is in canonical form (definition of canonical form), $(\Sigma x{:}B_1.\, B_2)^\ominus = \Sigma x{:}A_1.\, A_2$, and $\Gamma^\oplus \vdash A_1^{\oplus,\Gamma^\oplus} :: \Omega$ (DECL-T and Lemma 6.3.5) $\Rightarrow \Gamma^\oplus \vdash A_1^{\oplus,\Gamma^\oplus} = B_1 :: \Omega$ (Theorem 9.1.8) $\Rightarrow \Gamma^\oplus, x{:}B_1 \vdash B_2 :: \Omega$, (Theorem 7.6.9 and Theorem 7.6.10) $\Rightarrow \Gamma^\oplus \vdash \Sigma x{:}B_1.\, B_2 :: \Omega$ (C-DSUM) $\Rightarrow \Sigma x{:}A_1.\, A_2 \in C_\Gamma$. □

The following lemma allows weakening the assignment a semi-canonical type is relative to:

Lemma 9.5.5 (Weakening)
If $A \in C_{\Gamma_1;\Gamma_3}$, $\vdash \Gamma_1;\Gamma_2$ valid, and $dom(\Gamma_2) \cap dom(\Gamma_3) = \emptyset$ then $A \in C_{\Gamma_1;\Gamma_2;\Gamma_3}$.

Proof: By the definition of $A \in C_{\Gamma_1;\Gamma_3}$, $\exists B$ such that $B^\ominus = A$, $\Gamma_1^\oplus;\Gamma_3^{\oplus,\Gamma_1^\oplus} \vdash B :: \Omega$, and B is in canonical form. By Theorem 9.3.3 and Lemma 9.2.9, $\vdash \Gamma_1^\oplus;\Gamma_2^{\oplus,\Gamma_1^\oplus}$ valid $\Rightarrow \Gamma^\oplus;\Gamma_2^{\oplus,\Gamma_1^\oplus};\Gamma_3^{\oplus,\Gamma_1^\oplus} \vdash B :: \Omega$ (Lemma 9.2.10 and Theorem 6.8.10). By Lemma 9.2.11,

$,_3^{\oplus,_1^{\oplus}} = ,_3^{\oplus,_1;,_2^{\oplus,_1^{\oplus}}} \Rightarrow (,_1;,_2;,_3)^{\oplus} \vdash B :: \Omega \Rightarrow A \in C_{,_1;,_2;,_3}$ (by definition). □

The following propositions are transfered from Sections 8.4 and 8.6 using the results from Sections 9.1 to 9.3:

Lemma 9.5.6 *If $A \in C_{,}$, $, \vdash x\rho \Rightarrow A$, and $\mathcal{S}(A, x\rho, \rho')$ exists then $\mathcal{S}(A, x\rho, \rho') \in C_{,}$.*

Proof: By the definition of $A \in C_{,}$, there exists a tagged constructor B such that $B^{\ominus} = A$, $,^{\oplus} \vdash B :: \Omega$, and B is in canonical form. By Lemma 9.1.4, $\mathcal{S}(B, x\rho, \rho')$ exists. By Theorem 9.3.3, $,^{\oplus} \vdash x\rho \Rightarrow A^{\oplus,^{\oplus}} \Rightarrow ,^{\oplus} \vdash x\rho \Rightarrow B$ (P-MOVE, E-SYM, and Lemma 9.5.2) $\Rightarrow ,^{\oplus} \vdash \mathcal{S}(B, x\rho, \rho') :: \Omega$ (Theorems 7.6.13 and7.6.14). By Lemma 9.1.3, $\mathcal{S}(B, x\rho, \rho')^{\ominus} = \mathcal{S}(B^{\ominus}, x\rho, \rho') = \mathcal{S}(A, x\rho, \rho')$. By Lemma 8.4.6, $\mathcal{S}(B, x\rho, \rho')$ is in canonical form. Hence, $\mathcal{S}(A, x\rho, \rho') \in C_{,}$. □

Lemma 9.5.7 (Shape Pullback)
Suppose $, \vdash A_1 = A_2 :: \Omega$ and $A_1 \in C_{,}$. Then:

1. *If $A_2 = \alpha$, $A_2 =$ **ref**, $A_2 =$ **rec**, or $A_2 = \lt K\gt$ then $A_1 = A_2$.*

2. *If $A_2 = \Pi x{:}A_2'.\, A_2''$ then $\exists A_1', A_1''.\ A_1 = \Pi x{:}A_1'.\, A_1''$.*

3. *If $A_2 = \Sigma x{:}A_2'.\, A_2''$ then $\exists A_1', A_1''.\ A_1 = \Sigma x{:}A_1'.\, A_1''$.*

4. *If $A_2 = \lt{=}A_2'{::}K\gt$ then $\exists A_1'.\ A_1 = \lt{=}A_1'{::}K\gt$.*

5. *If $A_2 = \mathbf{ref}\, A_2'$ then $\exists A_1'.\ A_1 = \mathbf{ref}\, A_1'$.*

6. *If $A_2 = \mathbf{rec}\, A_2'$ then $\exists A_1'.\ A_1 = \mathbf{rec}\, A_1'$.*

Proof: By Theorem 9.3.3, $,^{\oplus} \vdash A_1^{\oplus,^{\oplus}} = A_2^{\oplus,^{\oplus}} :: \Omega$. By the definition of $A_1 \in C_{,}$, there exists a tagged constructor B such that $B^{\ominus} = A_1$ and B is in canonical form. By Lemma 9.5.2, $,^{\oplus} \vdash B = A_1^{\oplus,^{\oplus}} :: \Omega \Rightarrow ,^{\oplus} \vdash B = A_2^{\oplus,^{\oplus}} :: \Omega$ (E-TRAN). The desired results follows from Lemma 8.4.10 and the fact that the form of a constructor is unaffected by stamping and tag removal. □

Lemma 9.5.8 (Pullback)
If $A \in C_{,}$, $, \vdash A = A' :: \Omega$, $, \vdash x\rho \Rightarrow A'$, and $\mathcal{S}(A', x\rho, \rho')$ exists then $, \vdash \mathcal{S}(A, x\rho, \rho') = \mathcal{S}(A', x\rho, \rho') :: \Omega$.

Proof: By Theorem 9.3.3, $,^{\oplus} \vdash A^{\oplus_{,}^{\oplus}} = A'^{\oplus_{,}^{\oplus}} :: \Omega$ and $,^{\oplus} \vdash x\rho \Rightarrow A'^{\oplus_{,}^{\oplus}}$. By Lemmas 9.1.4 and 9.2.3, $\mathcal{S}(A'^{\oplus_{,}^{\oplus}}, x\rho, \rho')$ exists. Hence by Theorems 7.6.13 and7.6.14, $,^{\oplus} \vdash \mathcal{S}(A'^{\oplus_{,}^{\oplus}}, x\rho, \rho') :: \Omega$. By the definition of $A \in C_{,}$, there exists a tagged constructor B such that $B^{\ominus} = A$, $,^{\oplus} \vdash B :: \Omega$, and B is in canonical form. Hence, by Lemma 9.5.2, $,^{\oplus} \vdash B = A^{\oplus_{,}^{\oplus}} :: \Omega \Rightarrow ,^{\oplus} \vdash B = A'^{\oplus_{,}^{\oplus}} :: \Omega$ (E-TRAN). Hence by Lemma 8.4.12, $,^{\oplus} \vdash \mathcal{S}(B, x\rho, \rho') = \mathcal{S}(A'^{\oplus_{,}^{\oplus}}, x\rho, \rho') :: \Omega \Rightarrow , \vdash \mathcal{S}(B^{\ominus}, x\rho, \rho') = \mathcal{S}(A', x\rho, \rho') :: \Omega$ (Theorem 9.3.1 and Lemma 9.2.3) $\Rightarrow , \vdash \mathcal{S}(A, x\rho, \rho') = \mathcal{S}(A', x\rho, \rho') :: \Omega$. □

Lemma 9.5.9 *If* $,_1, x{:}A_1; ,_2 \vdash x\rho \Rightarrow A$ *and* $A_1 \in C_{,_1}$ *then* $,_1, x{:}A_1; ,_2 \vdash A = \mathcal{S}(A_1, x, \rho) :: \Omega$ *and* $\mathcal{S}(A_1, x, \rho) \in C_{,_1, x:A_1; ,_2}$.

Proof: Let $, = ,_1, x{:}A_1; ,_2$. By Theorem 9.3.3 and Lemma 9.2.9, $,_1^{\oplus}, x{:}A_1^{\oplus_{,_1}^{\oplus}}; ,_2^{\oplus(,_1, x:A_1)^{\oplus}} \vdash x\rho \Rightarrow A^{\oplus_{,}^{\oplus}}$. By the definition of $A_1 \in C_{,_1}$ and Lemma 9.5.2, $\exists B.\ B^{\ominus} = A_1$, B in canonical form, and $,_1^{\oplus} \vdash B = A_1^{\oplus_{,_1}^{\oplus}} :: \Omega$
$\Rightarrow ,_1^{\oplus}, x{:}B; ,_2^{\oplus(,_1, x:A_1)^{\oplus}} \vdash x\rho \Rightarrow A^{\oplus_{,}^{\oplus}}$ (Theorems 7.6.9 and 7.6.10).

Hence, by Lemmas 6.8.11 and 8.4.13, $,_1^{\oplus}, x{:}B; ,_2^{\oplus(,_1, x:A_1)^{\oplus}} \vdash A^{\oplus_{,}^{\oplus}} = \mathcal{S}(B, x, \rho) :: \Omega$ and $\mathcal{S}(B, x, \rho)$ is in canonical form $\Rightarrow ,^{\oplus} \vdash A^{\oplus_{,}^{\oplus}} = \mathcal{S}(B, x, \rho) :: \Omega$ (Theorems 7.6.9 and 7.6.10) $\Rightarrow ,^{\oplus} \vdash \mathcal{S}(B, x, \rho) :: \Omega$ (Theorem 7.6.10) $\Rightarrow \mathcal{S}(A_1, x, \rho) \in C_{,}$ (definition of a semi-canonical type and Lemma 9.1.3). By Theorem 9.3.1, Lemma 9.1.3, and Lemma 9.2.3, $, \vdash A = \mathcal{S}(A_1, x, \rho) :: \Omega$. □

Theorem 9.5.10 (Computability of semi-canonical types) *If* $, \vdash A :: \Omega$ *then a constructor* A' *is (recursively) computable such that* $, \vdash A = A' :: \Omega$ *and* $A' \in C_{,}$.

Proof: By Theorem 9.3.3, $,^{\oplus} \vdash A^{\oplus_{,}^{\oplus}} :: \Omega$. By Theorem 8.6.12, we can compute a tagged constructor B such that $,^{\oplus} \vdash A^{\oplus_{,}^{\oplus}} = B :: \Omega$ and B is in canonical form $\Rightarrow ,^{\oplus} \vdash B :: \Omega$ (Theorem 6.3.5) and $, \vdash A = B^{\ominus} :: \Omega$ (Theorem 9.3.1 and Lemma 9.2.3) $\Rightarrow B^{\ominus} \in C_{,}$. (Note that tag removal is computable; this is easily proved by structural induction on the item undergoing tag removal.) □

The last two lemmas of this section formalize when $x!$ is in semi-canonical form and that semi-canonical types for a given type under a given assignment are unique:

Lemma 9.5.11 *If* $, \vdash x! :: \Omega$ *and* $,(x) = <\Omega>$ *then* $x! \in C_{,}$.

Proof: By Lemma 9.2.4, $(x{:}{<}\Omega{>})^{\oplus,\ ^{\oplus}} \in\ ,^{\oplus} \Rightarrow\ ,^{\oplus}(x) = {<}\Omega{>}$ (Lemma 6.3.5, Theorem 9.3.3, and Lemma 9.4.2) $\Rightarrow\ ,^{\oplus} \vdash x_{<\Omega>}! :: \Omega$ (Theorem 9.3.3). By the definition of canonical form, $x_{<\Omega>}!$ is in canonical form $\Rightarrow x! \in C_{,}$ (definition of semi-canonical types). □

Lemma 9.5.12 *If $A_1 \in C_{,}\ ,\ A_2 \in C_{,}\ ,$ and $,\ \vdash A_1 = A_2 :: \Omega$ then $A_1 = A_2$.*

Proof: By the definition of $A_i \in C_{,}\ ,\ \exists B_i$ such that $B_i^{\ominus} = A,\ ,^{\oplus} \vdash B_i :: \Omega$, and B_i is in canonical form $\Rightarrow\ ,^{\oplus} \vdash B_i = A_i^{\oplus,\ ^{\oplus}} :: \Omega$ (Lemma 9.5.2). By Theorem 9.3.3, $,^{\oplus} \vdash A_1^{\oplus,\ ^{\oplus}} = A_2^{\oplus,\ ^{\oplus}} :: \Omega \Rightarrow\ ,^{\oplus} \vdash B_1 = B_2 :: \Omega$ (E-SYM and E-TRAN) $\Rightarrow B_1 \downarrow^{*} B_2$ (Theorem 7.8.5) $\Rightarrow B_1 = B_2$ (Lemma 8.1.3 and the definition of canonical form) $\Rightarrow B_1^{\ominus} = B_2^{\ominus}$ $\Rightarrow A_1 = A_2$ □

9.6 Strengthening

In this section I use the results about semi-canonical types to prove strengthening (removal of unreferenced declarations from an assignment) for the valid assignment, valid constructor, and equal constructor judgments. I previously showed strengthening by removing constructor declarations (Theorem 7.6.2 and Corollary 7.6.4); here I show strengthening by removing term variable declarations:

Theorem 9.6.1 (Strengthening) *Suppose $x \notin FTV(,\ ') \cup FTV(A)$. Then:*

1. *If $\vdash\ ,\ ,x{:}A';\ ,\ '$ valid then $\vdash\ ,\ ;\ ,\ '$ valid.*
2. *If $,\ ,x{:}A';\ ,\ ' \vdash A :: K$ then $,\ ;\ ,\ ' \vdash A :: K$.*

Proof: By simultaneous structural induction on the typing derivations. The only interesting cases are:

C-EXT-O: Given $,\ ,x{:}A';\ ,\ ' \vdash x'\rho! :: K$, $x' \neq x$, derived via rule C-EXT-O from $,\ ,x{:}A';\ ,\ ' \vdash x'\rho \Rightarrow {<}K{>}$.

Casing on the order of x and x' in $,\ ,x{:}A';\ ,\ '$ (Theorem 6.5.6 and Lemma 9.4.2):

- x occurs first:
  $\exists,\ _2,\ ,\ '_2.\ ,\ _2, x'{:}A_1;\ ,\ _2 =\ ,\ '$ where $A_1 = (,\ ,x{:}A';\ ,\ ')(x')$.
  Let $,\ _1 =\ ,\ ,x{:}A';\ ,\ _2,\ ,\ '_1 =\ ,\ '_2,\ ,\ _0 =\ ,\ ;\ ,\ _2$, and $,\ '_0 =\ ,\ '_2$.

- x' occurs first:
  $\exists , _2, , '_2 .\ , _2, x'{:}A_1, , '_2 = ,$ where $A_1 = (, , x{:}A'; , ')(x')$.
  Let $, _1 = , _2, , '_1 = , '_2, x{:}A'; , ', , _0 = , _2$ and $, '_0 = , '_2; , '$.

Either way, $, _1, x'{:}A_1, , '_1 = , , x{:}A'; , '$ and $, _0, x'{:}A_1, , '_0 = , ; , '$

$\Rightarrow\ \vdash , , x{:}A'; , '$ valid, $\vdash , _1$ valid, and $, _1 \vdash A_1 :: \Omega$ are sub-derivations of the given derivation (repeated use of Lemma 6.3.5 and DECL-T).

$\Rightarrow\ \vdash , ; , '$ valid and $, _0 \vdash A_1 :: \Omega$ (induction hypothesis)

By Theorem 9.5.10, $\exists A_2 .\ , _0 \vdash A_1 = A_2 :: \Omega$ and $A_2 \in C_{, _0}$

$\Rightarrow , _1 \vdash A_1 = A_2 :: \Omega$ and $A_2 \in C_{, _1}$ (Lemma 6.5.10, Theorem 9.4.1, and Lemma 9.5.5)

$\Rightarrow\ , _1, x'{:}A_2, , '_1 \vdash x'\rho \Rightarrow {<}K{>}$ (Theorem 9.4.6)

By Lemma 9.5.9 then, $, _1, x'{:}A_2; , '_1 \vdash {<}K{>} = \mathcal{S}(A_2, x', \rho) :: \Omega$ and $\mathcal{S}(A_2, x', \rho) \in C_{, _1, x'{:}A_2; , '_1} \Rightarrow \mathcal{S}(A_2, x', \rho)$ has form ${<}K{>}$ (E-SYM and Lemma 9.5.7). By P-INIT, $, ; , ' \vdash x' \Rightarrow A_1\ \Rightarrow\ , ; , ' \vdash x'\rho \Rightarrow \mathcal{S}(A_2, x', \rho)$ (Theorem 9.4.1 and P-MOVE) $\Rightarrow\ , ; , ' \vdash x'\rho \Rightarrow {<}K{>}\ \Rightarrow\ , ; , ' \vdash x'\rho! :: K$ (C-EXT-O).

C-EXT-T: Handled the same as the C-EXT-O case except that the form in question is ${<}{=}A''{::}K{>}$.

□

Corollary 9.6.2 (Strengthening)
Suppose $x \notin FTV(, ') \cup FTV(A_1) \cup FTV(A_2)$. Then, if $, , x{:}A'; , ' \vdash A_1 = A_2 :: K$, then $, ; , ' \vdash A_1 = A_2 :: K$.

Proof: By Theorems 9.4.3 and 9.6.1, $, ; , ' \vdash A_i :: K$. The desired result then follows from Theorem 9.4.8. □

A useful corollary is that replacing a type with an equal semi-canonical type never introduces new free term variables:

Corollary 9.6.3 (Non-Free Variable Pullback)
If $, , x{:}A \vdash A_1 = A_2 :: \Omega$, $A_1 \in C_{, , x:A}$, $x \notin FTV(A_2)$ then $x \notin FTV(A_1)$.

Proof: By Theorem 9.4.3, $, , x{:}A \vdash A_2 :: \Omega \Rightarrow , \ \vdash A_2 :: \Omega$ (Theorem 9.6.1) and $\vdash , , x{:}A$ valid (Lemma 6.3.5) $\Rightarrow x \notin \mathrm{dom}(,)$ (DECL-T). By Theorem 9.3.3,

$$, ^{\oplus}, x{:}A^{\oplus, ^{\oplus}} \vdash A_1^{\oplus, ^{\oplus}, x:A^{\oplus, ^{\oplus}}} = A_2^{\oplus, ^{\oplus}, x:A^{\oplus, ^{\oplus}}} :: \Omega \text{ and } , ^{\oplus} \vdash A_2^{\oplus, ^{\oplus}} :: \Omega$$

$$\Rightarrow\ , ^{\oplus}, x{:}A^{\oplus, ^{\oplus}} \vdash A_1^{\oplus, ^{\oplus}, x:A^{\oplus, ^{\oplus}}} = A_2^{\oplus, ^{\oplus}} :: \Omega \text{ (inspection of the definition of stamping).}$$

By the definition of $A_1 \in C_{, ,x:A}$, there exists a tagged constructor B such that: $B^{\ominus} = A_1$, $,^{\oplus}, x{:}A^{\oplus_{,^{\oplus}}} \vdash B :: \Omega$, and B is in canonical form $\Rightarrow$ $,^{\oplus}, x{:}A^{\oplus_{,^{\oplus}}} \vdash B = A_2^{\oplus_{,^{\oplus}}} :: \Omega$ (Lemma 9.5.2 and E-TRAN) $\Rightarrow$ $,^{\oplus} \vdash B = A_2^{\oplus_{,^{\oplus}}} :: \Omega$ (Corollary 8.4.11) $\Rightarrow$ $, \vdash A_1 = A_2 :: \Omega$ (Theorem 9.3.1 and Lemma 9.2.3) $\Rightarrow$ $, \vdash A_1 :: \Omega$ (Theorem 9.4.3) $\Rightarrow$ $\mathrm{FTV}(A_1) \subseteq \mathrm{dom}(,)$ (Theorem 6.5.6) $\Rightarrow$ $x \notin \mathrm{FTV}(A_1)$. □

9.7 Alternatives

I have now finished the part of the proofs devoted to constructor validity and equality. Once I was able to reach this point in the proofs, I did not have to make any more changes in this part of the system. I had to do considerable backtracking in the later parts of the proofs, but I never had to back up past this point again.

Accordingly, this is a good point to stop and consider alternative formulations of the system. The following suggestions are what I would try if I were to start the system and proofs over again from scratch. Be warned that these are highly speculative ideas and that although they are based on my experience, it is quite possible that they may be found to be unusable only after months of work.

I think the biggest source of complexity in the proofs is the mutual dependency between the valid constructor and equal constructor judgments. I suggest that this cycle be broken by dropping the current requirement that equations may involve only valid constructors. Let equal constructors under , be defined as two constructors that have the same kind under , and that are convertible via a rewriting relation on (possibly invalid) constructors. Under an assignment approach (cf. Section 6.2), this might result in the following rule for equality:

$$\frac{, \vdash A_1 :: K \qquad , \vdash A_2 :: K \qquad , , x{:}A_1 \approx , , x{:}A_2 \qquad x \notin \mathrm{dom}(,)}{, \vdash A_1 = A_2 :: K} \quad \text{(EQUAL)}$$

(Note that assignments as well as constructors have to be converted under the assignment approach because rewriting depends on the current assignment.)

The conversion relation can then be defined directly using a rewriting relation without any reference to the valid or equal constructor judgments. By adding the following rule, the rewriting relation can be made confluent on (possibly invalid) constructors even in the presence of both β and η:

$$\frac{A \rightarrow_{, ,\alpha::\Omega} A'}{\lambda\alpha{::}K.\, A \rightarrow_{,} \lambda\alpha{::}\Omega.\, A'} \quad \text{(ARG-KIND)}$$

This addition avoids the need to use erasure or subject reduction in order to prove confluence while preserving all the needed properties of the equality relation. This approach also avoids the need to prove that the rewriting relation implements a separately defined equality relation.

Ideally then, the rewriting relation could be introduced first. Only after its properties had been proved (confluence, kind preservation, shape preservation, etc.), the valid constructor and equal constructor judgments could be introduced. This arrangement would allow breaking up the first part of the proof into two mostly separate parts.

The ARG-KIND rule does complicate normalization somewhat. Subject reduction will only hold if the ARG-KIND rule is excluded. This means that normalization will have to proceed in two steps: First, normalize as before under the rewriting relation modulo the ARG-KIND rule. Second, use ARG-KIND to change all argument kinds to Ω. This last step cannot enable any new reductions so we will have a normal form.

I believe the approach sketched above should be substantially simpler than the one I used. I suggest the use of the assignment approach rather than the tagged one because in retrospect it seems simpler.

Chapter 10

Subtyping

In this chapter I introduce the subtyping relation ($, \vdash A \leq A'$) and prove the major results about it: subtypes' shapes are related, judgments may be weakened by replacing types in assignments by subtypes, subtyping acts in a component-wise manner, a semi-decision procedure for deciding subtyping exists, and subtyping is undecidable. The replacement by a subtype result will be crucial in proving soundness for the kernel system in the next chapter because it shows that values of a subtype can be substituted where values of a supertype are expected.

The replacement by a subtype result is hard to prove, however. One problem is that proving it directly requires knowing that subtyping acts in a component-wise manner; unfortunately, proving that subtyping acts in a component-wise manner requires the replacement by a subtype result in order to handle transitivity,[1] resulting in a cycle.

I shall avoid this problem by introducing a *one-step subtyping* relation ($, \vdash A \lhd A'$). The one-step subtyping relation is quite simple and only allows information about constructor components to be forgotten. It has no transitivity rule and does not allow rewriting via equality of constructors, thus avoiding the equality transitivity rule as well. Because of the lack of transitivity rules, it can be proved to act in a component-wise manner without using a replacement result. The lack of equality also simplifies relating the shapes of one-step subtypes.

The full subtyping relation can be constructed from the one-step subtyping relation and equality by defining A to be a subtype of A' under , iff you can get from A to A' by a series of applications of equality and one-step subtyping under , . For example, we might have $, \vdash A \leq A'$ by $, \vdash A = A_1 :: \Omega$, $, \vdash A_1 \lhd A_2$, $, \vdash A_2 = A_3 :: \Omega$, and $, \vdash A_3 \lhd A'$.

Using this definition of subtyping, the replacement by a subtype result can be proved as follows: First, prove that types in assignments can be replaced by one-step subtypes using the fact that one-step subtyping acts in a component-wise manner. Second, combine

[1]This fact can be seen in the use of Theorem 10.4.7 in the second to last step of Lemma 10.5.1.

that result with the replacement by an equal type result (Theorem 9.4.6) to prove the desired result. (The key idea for the second part is that you can always replace a type by a subtype by means of a sequence of replacements by equal and one-step subtypes; the sequence needed can be determined from the subtyping derivation.)

10.1 One-Step Subtyping

The rules for the one-step subtyping relation follow. Note that aside from the O-FORGET rule, the rules act either in a component-wise manner or leave the constructor unchanged.

Definition 10.1.1 (One-Step Subtyping Rules)

$$\frac{,\ \vdash A :: \Omega}{,\ \vdash A \lhd A} \qquad \text{(O-REFL)}$$

$$\frac{\begin{array}{c},\ \vdash A_1' \lhd A_1 \\ ,\ ,x{:}A_1' \vdash A_2 \lhd A_2' \qquad ,\ ,x{:}A_1 \vdash A_2 :: \Omega\end{array}}{,\ \vdash \Pi x{:}A_1.\, A_2 \lhd \Pi x{:}A_1'.\, A_2'} \qquad \text{(O-DFUN)}$$

$$\frac{\begin{array}{c},\ \vdash A_1 \lhd A_1' \\ ,\ ,x{:}A_1 \vdash A_2 \lhd A_2' \qquad ,\ ,x{:}A_1' \vdash A_2' :: \Omega\end{array}}{,\ \vdash \Sigma x{:}A_1.\, A_2 \lhd \Sigma x{:}A_1'.\, A_2'} \qquad \text{(O-DSUM)}$$

$$\frac{,\ \vdash A :: K}{,\ \vdash \,<\!=\!A{::}K\!>\, \lhd \,<\!K\!>} \qquad \text{(O-FORGET)}$$

The usual propositions on judgments also hold for one-step subtyping:

Lemma 10.1.2 (Validity of one-step subtypes)
If $,\ \vdash A \lhd A'$ *then* $,\ \vdash A :: \Omega$ *and* $,\ \vdash A' :: \Omega$.

Theorem 10.1.3 (Weakening)
Suppose $\vdash ,\ ;,'$ *valid and* $dom(,') \cap dom(,'') = \emptyset$. *Then if* $,\ ;,'' \vdash A_1 \lhd A_2$ *then* $,\ ;,';,'' \vdash A_1 \lhd A_2$.

Proof: By structural induction on the derivation of $,\ ;,'' \vdash A_1 \lhd A_2$, using Theorem 9.4.1 as needed. □

Lemma 10.1.4 (Strengthening)
Suppose $x \notin FTV(,') \cup FTV(A_1) \cup FTV(A_2)$. *Then, if* $,\ ,x{:}A';,' \vdash A_1 \lhd A_2$, *then* $,\ ;,' \vdash A_1 \lhd A_2$.

Proof: By structural induction on the derivation using Theorem 9.6.1 as needed. $\Box$

Lemma 10.1.5 (Validity of place substitution)
Suppose $, \vdash x\rho \Rightarrow A'$. *Then, if* $, , x'{:}A'; ,' \vdash A_1 \lhd A_2$ *then* $, ; [x\rho/x'], ' \vdash [x\rho/x']A_1 \lhd [x\rho/x']A_2$.

Proof: Proved by structural induction on the derivation of $, , x'{:}A'; ,' \vdash A_1 \lhd A_2$ using Theorem 9.4.7 as needed. $\Box$

Lemma 10.1.6 (Replacement with an equal type)
Suppose $, \vdash A_1 = A_2 :: \Omega$. *Then, if* $, , x{:}A_1; ,' \vdash A \lhd A'$ *then* $, , x{:}A_2; ,' \vdash A \lhd A'$.

Proof: Proved by structural induction on the derivation of $, , x{:}A_1; ,' \vdash A \lhd A'$ using Theorem 9.4.6 as needed. $\Box$

Because one-step subtyping lacks any transitivity rules, it can easily be shown to act in a component-wise manner:

Lemma 10.1.7 (Component-wise one-step subtyping)

1. *If* $, \vdash \Pi x{:}A_1.\, A_2 \lhd \Pi x{:}A_1'.\, A_2'$, $x \notin dom(,)$, *then* $, \vdash A_1' \lhd A_1$ *and* $, , x{:}A_1' \vdash A_2 \lhd A_2'$.

2. *If* $, \vdash \Sigma x{:}A_1.\, A_2 \lhd \Sigma x{:}A_1'.\, A_2'$, $x \notin dom(,)$, *then* $, \vdash A_1 \lhd A_1'$ *and* $, , x{:}A_1 \vdash A_2 \lhd A_2'$.

Proof: Proved by structural induction on the derivations. The O-DFUN and O-DSUM cases are immediate. The only other possible case is O-REFL which is handled by Lemma 10.1.2 and C-DFUN/C-DSUM (to establish the validity of A_1 and A_2) followed by O-REFL. $\Box$

Using this result, I can prove that (single-step) selections can be pulled back across one-step subtyping:

Lemma 10.1.8 (Pullback)
If $, \vdash x\rho \Rightarrow A_1$ *and* $, \vdash A_1 \lhd \Sigma x'{:}A_2'.\, A_2''$ *then* $, \vdash \mathcal{S}(A_1, x\rho, .i) \lhd \mathcal{S}(\Sigma x'{:}A_2'.\, A_2'', x\rho, .i)$.

Proof: WLOG, let $x' \notin$ dom(,). Inspection of the rules for one-step subtyping shows that $, \vdash A_1 \lhd \Sigma x'{:}A_2'.\, A_2''$ must have been derived using either O-REFL or O-DSUM. Either way, WLOG, A_1 must have the form $\Sigma x'{:}A_1'.\, A_1''$ for some A_1' and A_1'' $\Rightarrow$ $, \vdash A_1' \lhd A_2'$ and $, , x'{:}A_1' \vdash A_1'' \lhd A_2''$ (Lemma 10.1.7) and $\mathcal{S}(A_1, x\rho, .1) = A_1'$ exists $\Rightarrow$ $, \vdash \mathcal{S}(A_1, x\rho, .1) \lhd \mathcal{S}(\Sigma x'{:}A_2'.\, A_2'', x\rho, .1)$.

By Theorem 9.4.4, E-REFL, and P-MOVE, $, \vdash x\rho.1 \Rightarrow A_1'$
$\Rightarrow , \vdash [x\rho.1/x']A_1'' \lhd [x\rho.1/x']A_2''$ (Lemma 10.1.5)
$\Rightarrow , \vdash \mathcal{S}(A_1, x\rho, .2) \lhd \mathcal{S}(\Sigma x'{:}A_2'.\, A_2'', x\rho, .2)$.
Thus, $, \vdash \mathcal{S}(A_1, x\rho, .i) \lhd \mathcal{S}(\Sigma x'{:}A_2'.\, A_2'', x\rho, .i)$. □

10.2 The Subtyping Relation

The rules for the full subtyping relation are as follows:

Definition 10.2.1 (Subtyping Rules)

$$\frac{, \vdash A = A' :: \Omega}{, \vdash A \leq A'} \qquad \text{(S-EQ)}$$

$$\frac{, \vdash A \lhd A'}{, \vdash A \leq A'} \qquad \text{(S-ONE)}$$

$$\frac{, \vdash A \leq A' \qquad , \vdash A' \leq A''}{, \vdash A \leq A''} \qquad \text{(S-TRAN)}$$

Again, the usual propositions on judgments hold:

Lemma 10.2.2 (Validity of subtypes)
If $, \vdash A \leq A'$ *then* $, \vdash A :: \Omega$ *and* $, \vdash A' :: \Omega$.

Theorem 10.2.3 (Weakening)
Suppose $\vdash , ; , '$ *valid and dom*$(, ') \cap$ *dom*$(, '') = \emptyset$. *Then if* $, ; , '' \vdash A_1 \leq A_2$ *then* $, ; , '; , '' \vdash A_1 \leq A_2$.

Proof: By structural induction on the derivation of $, ; , '' \vdash A_1 \leq A_2$, using Theorems 9.4.1 and 10.1.3 as needed. □

Theorem 10.2.4 (Strengthening)
Suppose $x \notin FTV(, ') \cup FTV(A_1) \cup FTV(A_2)$. *Then, if* $, , x{:}A'; , ' \vdash A_1 \leq A_2$, *then* $, ; , ' \vdash A_1 \leq A_2$.

Proof: By structural induction on the derivation using Lemma 10.1.4 and Corollary 9.6.2 as needed. $\Box$

Lemma 10.2.5 (Replacement with an equal type)
Suppose $, \vdash A_1 = A_2 :: \Omega$. *Then, if* $, , x{:}A_1; ,' \vdash A \leq A'$ *then* $, , x{:}A_2; ,' \vdash A \leq A'$.

Proof: Proved by structural induction on the derivation of $, , x{:}A_1; ,' \vdash A \leq A'$ using Theorem 9.4.6 and Lemma 10.1.6 as needed. $\Box$

10.3 Shapes

In this section I formalize the notion of constructor shapes and establish results about how the shapes of constructors related by the equality, one-step subtyping, and subtyping judgments are related. I shall use the following set of shapes to describe the structure of constructors:

Definition 10.3.1 (Shape syntax)

$$\textit{Shapes} \quad \chi \quad ::= \quad \textit{?} \mid \nu \mid \Pi \mid \Sigma \mid \lambda \mid {<}K{>} \mid {<}{=}{::}K{>} \mid \mathbf{rec} \mid \mathbf{ref} \mid \mathbf{recapp} \mid \mathbf{refapp}$$

The has-shape function ($\lceil \Leftrightarrow \rceil$) assigns these shapes to constructors in the following manner:

Definition 10.3.2 (Constructor shapes)

$$\begin{array}{rcll}
\lceil \alpha \rceil & = & \nu & \\
\lceil \Pi x{:}A.\, A' \rceil & = & \Pi & \\
\lceil \Sigma x{:}A.\, A' \rceil & = & \Sigma & \\
\lceil \lambda \alpha{::}K.\, A \rceil & = & \lambda & \\
\lceil \mathbf{rec}\, A' \rceil & = & \mathbf{recapp} & \\
\lceil \mathbf{ref}\, A' \rceil & = & \mathbf{refapp} & \\
\lceil A\, A' \rceil & = & \textit{?} & (A \notin \{\mathbf{rec}, \mathbf{ref}\}) \\
\lceil {<}K{>} \rceil & = & {<}K{>} & \\
\lceil {<}{=}A{::}K{>} \rceil & = & {<}{=}{::}K{>} & \\
\lceil x\rho! \rceil & = & \textit{?} & \\
\lceil \mathbf{rec} \rceil & = & \mathbf{rec} & \\
\lceil \mathbf{ref} \rceil & = & \mathbf{ref} &
\end{array}$$

I have chosen the set of shapes so that if the ? shape is considered a wildcard, equal to any other shape, then equal constructors always have equal shapes:

Lemma 10.3.3 (Shapes and equality)
Suppose , $\vdash A_1 = A_2 :: \Omega$. *Then one of the following is true:*

1. $\lceil A_1 \rceil =$ *?*
2. $\lceil A_2 \rceil =$ *?*
3. $\lceil A_1 \rceil = \lceil A_2 \rceil$

Proof: By Theorem 9.4.3, , $\vdash A_1 :: \Omega$ and , $\vdash A_2 :: \Omega \Rightarrow \lceil A_1 \rceil \neq \lambda$ and $\lceil A_2 \rceil \neq \lambda$ (inspection of the typing rules). The desired result then follows by Lemma 9.4.9. □

In order to describe how one-step subtyping interacts with shapes, it is useful to introduce the following ordering relation on shapes, which puts transparent type shapes before opaque type shapes that contain constructors of the same kind:

Definition 10.3.4 (Ordering of shapes)
We say that $\chi_1 \leq \chi_2$ *iff either of the following:*

1. $\chi_1 = \chi_2$
2. $\chi_1 = $ <=::K> *and* $\chi_2 = $ <K> *for some* K

Note that this ordering relation is transitive:

Lemma 10.3.5 (Transitivity)
If $\chi_1 \leq \chi_2$ *and* $\chi_2 \leq \chi_3$ *then* $\chi_1 \leq \chi_3$.

The shapes of constructors related by one-step subtyping are related by this ordering:

Lemma 10.3.6 (Shapes and one-step subtyping)
If , $\vdash A_1 \lhd A_2$ *then* $\lceil A_1 \rceil \leq \lceil A_2 \rceil$.

Proof: Proved by inspection of the one-step subtyping rules. □

If one-step subtyping is applied to a constructor with the ? shape, it acts as the identity relation:

Lemma 10.3.7 *If* , $\vdash A_1 \lhd A_2$ *and* $\lceil A_1 \rceil =$ *?* *or* $\lceil A_2 \rceil =$ *?* *then:*

1. $A_1 = A_2$
2. $, \vdash A_1 = A_2 :: \Omega$
3. $\lceil A_1 \rceil = \lceil A_2 \rceil = ?$

Proof: Inspection of the one-step subtyping rules shows that $, \vdash A_1 \lhd A_2$ must be derived using the O-REFL rule $\Rightarrow A_1 = A_2 \Rightarrow\ , \vdash A_1 = A_2 :: \Omega$ (Theorem 9.4.3 and E-REFL). □

When dealing with multiple equalities or one-step subtypings (as is the case when dealing with the full subtyping relation), it is useful to introduce the concept of a constructor's *intrinsic shape* under a given assignment ($\lceil \Leftrightarrow \rceil_{,}$). The intrinsic shape of a constructor is the most defined shape it can be put into using equality where ? is considered the least defined shape:

Definition 10.3.8 (Intrinsic constructor shapes)
If $, \vdash A :: K$ *then define* $\lceil A \rceil_{,}$ *as follows:*

1. *If* $\exists A'$ *such that* $, \vdash A = A' :: K$, $\lceil A' \rceil = \chi$, *and* $\chi \neq ?$ *then* $\lceil A \rceil_{,} = \chi$.
2. *Otherwise,* $\lceil A \rceil_{,} = ?$.

(Note that this definition is well defined because Lemma 10.3.3 and E-TRAN imply that case one cannot hold for two different shapes.)

When necessary, a type's intrinsic shape under a given assignment may be computed by first computing an equal semi-canonical type (via Theorem 9.5.10) then determining the resulting semi-canonical type's shape:

Lemma 10.3.9 *If* $A \in C_{,}$ *then* $\lceil A \rceil_{,} = \lceil A \rceil$.

Proof: Suppose not. Then by the definition of intrinsic shape and E-REFL, $\lceil A \rceil = ?$ and $\lceil A \rceil_{,} \neq ? \Rightarrow \exists A'$ such that $, \vdash A = A' :: K$, $\lceil A' \rceil = \lceil A \rceil_{,}$. By Lemma 9.5.2, $, \vdash A :: \Omega$. By Theorems 9.4.5 and 9.4.3, $K = \Omega$ and $, \vdash A' :: \Omega$. $\Rightarrow \lceil A' \rceil \neq \lambda$ (inspection of typing rules). By Lemma 9.5.7 then, $\lceil A \rceil = \lceil A' \rceil = \lceil A \rceil_{,}$. □

Equal constructors, of course, have the same intrinsic shape:

Lemma 10.3.10 *If* $, \vdash A_1 = A_2 :: K$ *then* $\lceil A_1 \rceil_{,} = \lceil A_2 \rceil_{,}$.

Proof: By E-TRAN, A_1 and A_2 are equal to exactly the same types under , . Hence, by the definition of intrinsic constructor shapes, their intrinsic constructor shapes must be the same. □

Both forms of subtyping relate types whose intrinsic shapes are related by the introduced ordering on shapes:

Lemma 10.3.11 *If* $, \vdash A_1 \lhd A_2$ *then* $\lceil A_1 \rceil_, \leq \lceil A_2 \rceil_, .$

Proof: Cases:

Case I: $\lceil A_2 \rceil = ?$
$\Rightarrow A_1 = A_2$ (Lemma 10.3.7) $\Rightarrow \lceil A_1 \rceil_, = \lceil A_2 \rceil_, \Rightarrow \lceil A_1 \rceil_, \leq \lceil A_2 \rceil_, .$

Case II: $\lceil A_2 \rceil \neq ?$
By Lemma 10.3.6, $\lceil A_1 \rceil \leq \lceil A_2 \rceil \Rightarrow \lceil A_1 \rceil \neq ?$ $(? \leq \chi \Rightarrow \chi = ?) \Rightarrow \lceil A_1 \rceil = \lceil A_1 \rceil_,$ and $\lceil A_2 \rceil = \lceil A_2 \rceil_,$ (Lemma 10.1.2, E-REFL, and the definitions of shape and intrinsic shape) $\Rightarrow \lceil A_1 \rceil_, \leq \lceil A_2 \rceil_, .$

□

Corollary 10.3.12 *If* $, \vdash A_1 \leq A_2$ *then* $\lceil A_1 \rceil_, \leq \lceil A_2 \rceil_, .$

Proof: Proved by structural induction on the derivation, using Lemma 10.3.11, Lemma 10.3.5, and the definition of intrinsic shape as needed. □

Subtyping collapses to equality when applied to types with certain intrinsic shapes:

Lemma 10.3.13 *If* $, \vdash A_1 \lhd A_2$, $\lceil A_1 \rceil_, = \lceil A_2 \rceil_,$ *, and* $\lceil A_2 \rceil_, \notin \{\Pi, \Sigma\}$ *then* $, \vdash A_1 = A_2 :: \Omega$

Proof: Lemma 10.3.7 handles the case where $\lceil A_1 \rceil = ?$ or $\lceil A_2 \rceil = ?$, so we can assume that $\lceil A_1 \rceil \neq ?$ and $\lceil A_2 \rceil \neq ? \Rightarrow \lceil A_1 \rceil = \lceil A_1 \rceil_, = \lceil A_2 \rceil_, = \lceil A_2 \rceil$ (definition of intrinsic shape) $\Rightarrow , \vdash A_1 \lhd A_2$ was derived using O-REFL (inspection of the one-step subtyping rules) $\Rightarrow A_1 = A_2 \Rightarrow , \vdash A_1 = A_2 :: \Omega$ (Theorem 9.4.3 and E-REFL). □

Corollary 10.3.14 *If* $, \vdash A_1 \leq A_2$, $\lceil A_1 \rceil_, = \lceil A_2 \rceil_,$ *, and* $\lceil A_2 \rceil_, \notin \{\Pi, \Sigma\}$ *then* $, \vdash A_1 = A_2 :: \Omega$

Proof: Inspection of the rules for subtyping shows that a subtyping derivation for $, \vdash A_1 \leq A_2$ can be regarded as a linear sequence of types starting with A_1 and ending with A_2 such that adjacent types are related either by equality or one-step subtyping (under assignment ,). By Lemma 10.3.11 and the definition of intrinsic shape, it follows that the intrinsic shape with respect to , of all the types in the sequence are the same. Hence, by Lemma 10.3.13 then, we can replace the one-step subtyping steps in the sequence by equalities. By repeatedly applying E-TRAN then, we get the desired result. □

I shall need the following lemma about shapes and one-step subtyping in order to prove the replacement by a subtype result:

Lemma 10.3.15 (Shape preservation)
Suppose $, \vdash A_1 \lhd A_2$ *and* $, \vdash A_2 = A_3 :: \Omega$. *Then:*

1. *If* $\lceil A_3 \rceil = \texttt{<}K\texttt{>}$ *then either* $, \vdash A_1 = A_3 :: \Omega$ *or* $\lceil A_1 \rceil = \texttt{<=::}K\texttt{>}$.

2. *If* $\lceil A_3 \rceil = \texttt{<=::}K\texttt{>}$ *then* $, \vdash A_1 = A_3 :: \Omega$.

Proof: If $\lceil A_2 \rceil = ?$ then, by Lemma 10.3.7 and E-TRAN, we have that $, \vdash A_1 = A_3 :: \Omega$ and are done. So, assume otherwise $\Rightarrow \lceil A_2 \rceil = \lceil A_3 \rceil$
(Lemma 10.3.3). By Lemma 10.3.6, $\lceil A_1 \rceil \leq \lceil A_2 \rceil$. There are two cases:

case I: Here $\lceil A_1 \rceil = \lceil A_2 \rceil \Rightarrow \lceil A_1 \rceil \neq ? \Rightarrow \lceil A_1 \rceil_{,} = \lceil A_1 \rceil$ and $\lceil A_2 \rceil_{,} = \lceil A_2 \rceil$ (Lemma 10.1.2, E-REFL, and definition of intrinsic shape)
$\Rightarrow\ , \vdash A_1 = A_2 :: \Omega$ (Lemma 10.3.13) $\Rightarrow\ , \vdash A_1 = A_3 :: \Omega$ (E-TRAN).

case II: Here $\lceil A_1 \rceil = \texttt{<=::}K\texttt{>}$ and $\lceil A_3 \rceil = \lceil A_2 \rceil = \texttt{<}K\texttt{>}$. We must proving part two here and are thus done.

□

10.4 Replacement by a Subtype

In this section I prove the key result that if you take a true judgment and replace a type in its assignment with a subtype, the resulting judgment is also true. As I discussed earlier, I shall do this by first proving a similar result using the one-step subtyping relation instead of the full subtyping relation.

The induction hypothesis required for proving this statement is quite tricky; I shall need a notion of constructor size that decreases when a constructor is replaced by a selection on that constructor using a non-empty path:

Definition 10.4.1 (Σ-size) *If A has form $\Sigma x{:}A_1.\,A_2$ then $\lfloor A \rfloor = \lfloor A_1 \rfloor + \lfloor A_0 \rfloor + 1$. Otherwise, $\lfloor A \rfloor = 0$.*

Lemma 10.4.2 (Σ-size properties)

1. $\lfloor A \rfloor \geq 0$
2. $\lfloor [x\rho/x']A \rfloor = \lfloor A \rfloor$
3. *If $\mathcal{S}(A, x\rho, .i)$ exists then $\lfloor \mathcal{S}(A, x\rho, .i) \rfloor < \lfloor A \rfloor$.*
4. *If $\mathcal{S}(A, x\rho, \rho')$, $\rho' \neq \epsilon$, exists then $\lfloor \mathcal{S}(A, x\rho, \rho') \rfloor < \lfloor A \rfloor$.*

Proof: Parts one and two are proved by structural induction on A. Part three follows from parts one and two and the definitions of selection and Σ-size. Part four follows from part three and Lemma 6.5.3. □

In order to prevent the proof from getting too complicated, I have (partially) pulled out the following lemma that is used to handle the induction case for the place lookup judgment. Preconditions one through four of the lemma describe a simplified version of that case. In particular, the lemma assumes that the selection is by exactly one step ($.i$); I shall remove this simplification shortly.

The fifth precondition represents the induction hypothesis of the replacement by a one-step subtype result that we are trying to prove. This precondition allows the lemma to use the replacement result with smaller (in the Σ-size sense) constructors. The lemma says that under these preconditions, we can either directly lookup the end type ($\mathcal{S}(A_2, x\rho, .i)$) or we can lookup a type (A_3') that we can convert via one-step subtyping ($, \vdash A_3' \lhd A_1'$) followed by equality ($, \vdash A_1' = \mathcal{S}(A_2, x\rho, .i) :: \Omega$) to the end type. In the second case, the one-step supertype Σ-size is such that we can use the replacement result with it.

Lemma 10.4.3 (Special pullback lemma I) *Suppose:*

1. $, \vdash x\rho \Rightarrow A_3$
2. $, \vdash A_3 \lhd A_1$
3. $, \vdash A_1 = A_2 :: \Omega$
4. *$\mathcal{S}(A_2, x\rho, .i)$ exists*
5. *$\forall x', A, A', A_1', A_2'$. If $, , x'{:}A \vdash A_1' = A_2' :: \Omega$, $, \vdash A' \lhd A$, $\lfloor A \rfloor < \lfloor A_1 \rfloor$ then $, , x'{:}A' \vdash A_1' = A_2' :: \Omega$.*

Then one of the following holds:

- $, \vdash x\rho.i \Rightarrow \mathcal{S}(A_2, x\rho, .i)$
- $\exists A_3', A_1'.\ , \vdash x\rho.i \Rightarrow A_3',\ , \vdash A_3' \lhd A_1',\ \lfloor A_1' \rfloor < \lfloor A_1 \rfloor$, *and* $, \vdash A_1' = \mathcal{S}(A_2, x\rho, .i) :: \Omega$

Proof: Since $\mathcal{S}(A_2, x\rho, .i)$ exists, $\exists x', A_2', A_2''.\ A_2 = \Sigma x'{:}A_2'.\ A_2''$ and $x' \notin \mathrm{dom}(,)$ By Lemma 10.3.3, we have the following cases:

- $\lceil A_1 \rceil = ? \Rightarrow\ , \vdash A_3 = A_1 :: \Omega$ (Lemma 10.3.7) $\Rightarrow\ , \vdash A_3 = A_2 :: \Omega$ (E-TRAN) $\Rightarrow$ $, \vdash x\rho.i \Rightarrow \mathcal{S}(A_2, x\rho, .i)$ (P-MOVE)
- $\lceil A_1 \rceil = \Sigma \Rightarrow \exists A_1', A_1''.\ A_1 = \Sigma x'{:}A_1'.\ A_1'' \Rightarrow\ , \vdash \Sigma x'{:}A_1'.\ A_1'' = \Sigma x'{:}A_2'.\ A_2'' :: \Omega$,
  $, \vdash \mathcal{S}(A_3, x\rho, .1) \lhd \mathcal{S}(\Sigma x'{:}A_1'.\ A_1'', x\rho, .1)$, and
  $, \vdash \mathcal{S}(A_3, x\rho, .2) \lhd \mathcal{S}(\Sigma x'{:}A_1'.\ A_1'', x\rho, .2)$. (Lemma 10.1.8)
  $\Rightarrow\ , \vdash A_1' = A_2' :: \Omega$ and $, , x'{:}A_1' \vdash A_1'' = A_2'' :: \Omega$ (Lemma 9.4.10) and
  $, \vdash \mathcal{S}(A_3, x\rho, .1) \lhd A_1' \Rightarrow\ , \vdash \mathcal{S}(A_1, x\rho, .1) = \mathcal{S}(A_2, x\rho, .1) :: \Omega$.

  By Lemma 10.4.2, $\lfloor A_1' \rfloor = \lfloor \mathcal{S}(A_1, x\rho, .1) \rfloor < \lfloor A_1 \rfloor$ and $\lfloor \mathcal{S}(A_1, x\rho, .2) \rfloor < \lfloor A_1 \rfloor$. Hence, by precondition number five, $, , x'{:}\mathcal{S}(A_3, x\rho, .1) \vdash A_1'' = A_2'' :: \Omega$.

  By Theorem 9.4.4, E-REFL, and P-MOVE, $, \vdash x\rho.1 \Rightarrow \mathcal{S}(A_3, x\rho, .1)$ and $, \vdash x\rho.2 \Rightarrow \mathcal{S}(A_3, x\rho, .2) \Rightarrow\ , \vdash [x\rho.1/x']A_1'' = [x\rho.1/x']A_2'' :: \Omega$ (Lemma 10.1.5) $\Rightarrow$ $, \vdash \mathcal{S}(A_1, x\rho, .2) = \mathcal{S}(A_2, x\rho, .2) :: \Omega$.

  Thus, $, \vdash x\rho.i \Rightarrow \mathcal{S}(A_3, x\rho, .i),\ , \vdash \mathcal{S}(A_3, x\rho, .i) \lhd \mathcal{S}(A_1, x\rho, .i)$, $\lfloor \mathcal{S}(A_1, x\rho, .i) \rfloor < \lfloor A_1 \rfloor$, and $, \vdash \mathcal{S}(A_1, x\rho, .i) = \mathcal{S}(A_2, x\rho, .i) :: \Omega$.

$\square$

By iterating the previous lemma, we can remove the restriction that the selection be by only one step (the statement of the lemma is otherwise unchanged):

Lemma 10.4.4 (Special pullback lemma II) *Suppose:*

1. $, \vdash x\rho \Rightarrow A_3$
2. $, \vdash A_3 \lhd A_1$
3. $, \vdash A_1 = A_2 :: \Omega$
4. $\mathcal{S}(A_2, x\rho, \rho')$ *exists*
5. $\forall x', A, A', A_1', A_2'.$ *If* $, , x'{:}A \vdash A_1' = A_2' :: \Omega,\ , \vdash A' \lhd A,\ \lfloor A \rfloor < \lfloor A_1 \rfloor$ *then* $, , x'{:}A' \vdash A_1' = A_2' :: \Omega$.

Then one of the following holds:

- $, \vdash x\rho\rho' \Rightarrow \mathcal{S}(A_2, x\rho, \rho')$
- $\exists A_3', A_1'.\ , \vdash x\rho\rho' \Rightarrow A_3'$, $, \vdash A_3' \lhd A_1'$, $\lfloor A_1' \rfloor < \lfloor A_1 \rfloor$, *and* $, \vdash A_1' = \mathcal{S}(A_2, x\rho, \rho') :: \Omega$

Proof: Proved by induction on the length of ρ'. Cases:

Zero: Here $\rho' = \epsilon \Rightarrow \mathcal{S}(A_2, x\rho, \rho') = A_2$. Let $A_3' = A_3$ and $A_1' = A_1$ and we are done.

Many: Here $\rho' = \rho''.i$. By Lemma 6.5.3, $\mathcal{S}(A_2, x\rho, \rho''.i) = \mathcal{S}(\mathcal{S}(A_2, x\rho, \rho''), x\rho\rho'', .i)$.

Hence, by the induction hypothesis, we have two cases:

- Here $, \vdash x\rho\rho'' \Rightarrow \mathcal{S}(A_2, x\rho, \rho'') \Rightarrow\ , \vdash x\rho\rho''.i \Rightarrow \mathcal{S}(A_2, x\rho, \rho''.i)$ (Theorem 9.4.4, E-REFL, and P-MOVE) $\Rightarrow\ , \vdash x\rho\rho' \Rightarrow \mathcal{S}(A_2, x\rho, \rho')$
- Here $\exists A_3'', A_1''.\ , \vdash x\rho\rho'' \Rightarrow A_3''$, $, \vdash A_3'' \lhd A_1''$, $\lfloor A_1'' \rfloor < \lfloor A_1 \rfloor$, and $, \vdash A_1'' = \mathcal{S}(A_2, x\rho, \rho'') :: \Omega$. Since $\lfloor A_1'' \rfloor < \lfloor A_1 \rfloor$, we can apply Lemma 10.4.3 to get: $\exists A_3', A_1'.\ , \vdash x\rho\rho''.i \Rightarrow A_3'$, $, \vdash A_3' \lhd A_1'$, $\lfloor A_1' \rfloor < \lfloor A_1'' \rfloor$, and $, \vdash A_1' = \mathcal{S}(\mathcal{S}(A_2, x\rho, \rho''), x\rho\rho'', .i) :: \Omega \Rightarrow\ , \vdash x\rho\rho''.i \Rightarrow A_3'$, $, \vdash A_3' \lhd A_1'$, $\lfloor A_1' \rfloor < \lfloor A_1 \rfloor$, and $, \vdash A_1' = \mathcal{S}(A_2, x\rho\rho'', \rho') :: \Omega$

□

Using the improved version of the lemma, I can now prove the desired replacement result:

Theorem 10.4.5 (Replacement with a one-step subtype)
If $, \vdash A_2 \lhd A_1$ *then:*

1. *If* $\vdash\ ,\ , x{:}A_1;\ ,'$ *valid then* $\vdash\ ,\ , x{:}A_2;\ ,'$ *valid.*
2. *If* $,\ , x{:}A_1;\ ,' \vdash A :: K$ *then* $,\ , x{:}A_2;\ ,' \vdash A :: K$.
3. *If* $,\ , x{:}A_1;\ ,' \vdash A = A' :: K$ *then* $,\ , x{:}A_2;\ ,' \vdash A = A' :: K$.
4. *If* $,\ , x{:}A_1;\ ,' \vdash x'\rho \Rightarrow A$, $x' \neq x$, *then* $,\ , x{:}A_2;\ ,' \vdash x'\rho \Rightarrow A$.
5. *If* $,\ , x{:}A_1;\ ,' \vdash x\rho \Rightarrow A$ *then one of:*
   - $,\ , x{:}A_2;\ ,' \vdash x\rho \Rightarrow A$
   - $\exists A_2', A_1'.\ ,\ , x{:}A_2;\ ,' \vdash x\rho \Rightarrow A_2'$, $,\ , x{:}A_2;\ ,' \vdash A_2' \lhd A_1'$, $\lfloor A_1' \rfloor \leq \lfloor A_1 \rfloor$, *and* $,\ , x{:}A_2;\ ,' \vdash A_1' = A :: \Omega$

Proof: Proved by simultaneous induction on all five parts; the metric used is based first on $\lfloor A_1 \rfloor$ and then on the structure of the derivations of the parts. Thus, we may recursively call ourselves either with a A_1 of smaller Σ-size or with the same A_1 and a sub-derivation of the original one. Interesting cases:

C-EXT-O II: Given $, , x{:}A_1; , ' \vdash x\rho! :: K$ derived via C-EXT-O from $, , x{:}A_1; , ' \vdash x\rho \Rightarrow \texttt{<}K\texttt{>}$. By the induction hypothesis, Theorem 9.4.4, O-REFL, and E-REFL, we have $\exists A_2', A_1'.\ , , x{:}A_2; , ' \vdash x\rho \Rightarrow A_2'$, $, , x{:}A_2; , ' \vdash A_2' \lhd A_1'$, and
$, , x{:}A_2; , ' \vdash A_1' = \texttt{<}K\texttt{>} :: \Omega \Rightarrow , , x{:}A_2; , ' \vdash A_2' = \texttt{<}K\texttt{>} :: \Omega$ or $\exists A_2''.\ A_2' = \texttt{<=}A_2''\texttt{::}K\texttt{>}$
(Lemma 10.3.15) $\Rightarrow , , x{:}A_2; , ' \vdash x\rho \Rightarrow \texttt{<}K\texttt{>}$ (P-MOVE) or
$, , x{:}A_2; , ' \vdash x\rho \Rightarrow \texttt{<=}A_2''\texttt{::}K\texttt{>} \Rightarrow , , x{:}A_2; , ' \vdash x\rho! :: K$ (C-EXT-O or C-EXT-T).

C-EXT-T II: Given $, , x{:}A_1; , ' \vdash x\rho! :: K$ derived via C-EXT-T from
$, , x{:}A_1; , ' \vdash x\rho \Rightarrow \texttt{<=}A\texttt{::}K\texttt{>}$ By the induction hypothesis, Theorem 9.4.4, O-REFL, and E-REFL, we have $\exists A_2', A_1'.\ , , x{:}A_2; , ' \vdash x\rho \Rightarrow A_2'$, $, , x{:}A_2; , ' \vdash A_2' \lhd A_1'$, and
$, , x{:}A_2; , ' \vdash A_1' = \texttt{<=}A\texttt{::}K\texttt{>} :: \Omega \Rightarrow , , x{:}A_2; , ' \vdash A_2' = \texttt{<=}A\texttt{::}K\texttt{>} :: \Omega$
(Lemma 10.3.15) $\Rightarrow , , x{:}A_2; , ' \vdash x\rho \Rightarrow \texttt{<=}A\texttt{::}K\texttt{>}$ (P-MOVE) $\Rightarrow , , x{:}A_2; , ' \vdash x\rho! :: K$
(C-EXT-T).

E-ABBREV II: Given $, , x{:}A_1; , ' \vdash x\rho! = A' :: K$ derived via E-ABBREV from
$, , x{:}A_1; , ' \vdash x\rho! :: K$, $, , x{:}A_1; , ' \vdash x\rho \Rightarrow A$, and
$, , x{:}A_1; , ' \vdash A = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega$ By the induction hypothesis,
Theorem 9.4.4, O-REFL, and E-REFL, we have $, , x{:}A_2; , ' \vdash x\rho! :: K$,
$, , x{:}A_2; , ' \vdash A = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega$, and $\exists A_2', A_1'.\ , , x{:}A_2; , ' \vdash x\rho \Rightarrow A_2'$,
$, , x{:}A_2; , ' \vdash A_2' \lhd A_1'$, and $, , x{:}A_2; , ' \vdash A_1' = A :: \Omega$
$\Rightarrow , , x{:}A_2; , ' \vdash A_1' = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega$ (E-TRAN)
$\Rightarrow , , x{:}A_2; , ' \vdash A_2' = \texttt{<=}A'\texttt{::}K'\texttt{>} :: \Omega$ (Lemma 10.3.15)
$\Rightarrow , , x{:}A_2; , ' \vdash x\rho \Rightarrow A$ (P-MOVE, E-SYM, and E-TRAN)
$\Rightarrow , , x{:}A_2; , ' \vdash x\rho! = A' :: K$ (E-ABBREV).

P-INIT I: Given $, , x{:}A_1; , ' \vdash x' \Rightarrow A$, $x' \neq x$, derived via P-INIT from $\vdash , , x{:}A_1; , '$ valid and $x'{:}A \in , , x{:}A_1; , ' \Rightarrow x'{:}A \in , ; , '$ and $\vdash , , x{:}A_2; , '$ valid (induction hypothesis) $\Rightarrow$ $x'{:}A \in , , x{:}A_2; , ' \Rightarrow , , x{:}A_2; , ' \vdash x' \Rightarrow A$ (P-INIT).

P-INIT II: Given $, , x{:}A_1; , ' \vdash x \Rightarrow A$ derived via P-INIT from $\vdash , , x{:}A_1; , '$ valid and $x{:}A \in , , x{:}A_1; , ' \Rightarrow \vdash , , x{:}A_2; , '$ valid (induction hypothesis) and $A = A_1$ (Lemma 9.4.2) $\Rightarrow , , x{:}A_2; , ' \vdash x \Rightarrow A_2$ (P-INIT) and $, , x{:}A_2; , ' \vdash A_2 \lhd A_1$ and $, , x{:}A_2; , ' \vdash A_1 :: \Omega$ (Lemma 10.1.2 and Theorem 9.4.1) $\Rightarrow , , x{:}A_2; , ' \vdash A_1 = A_1 :: \Omega$ (E-REFL) Let $A_2' = A$ and $A_1' = A_1$ and we are done.

P-MOVE II: Given $, , x{:}A_1; , ' \vdash x\rho\rho' \Rightarrow A''$ derived via P-MOVE from $, , x{:}A_1; , ' \vdash x\rho \Rightarrow A$, $, , x{:}A_1; , ' \vdash A = A' :: \Omega$, and $A'' = \mathcal{S}(A', x\rho, \rho')$. Applying the induction hypothesis gives us that $, , x{:}A_2; , ' \vdash A = A' :: \Omega$ and that one of the following cases holds:

– Here $, \, , x{:}A_2; , ' \vdash x\rho \Rightarrow A \Rightarrow \, , \, , x{:}A_2; , ' \vdash x\rho\rho' \Rightarrow A''$ (P-MOVE).

– Here $\exists A_2'', A_1''. \, , \, , x{:}A_2; , ' \vdash x\rho \Rightarrow A_2'', \, , \, , x{:}A_2; , ' \vdash A_2'' \lhd A_1''$, $\lfloor A_1'' \rfloor \leq \lfloor A_1 \rfloor$, and $, \, , x{:}A_2; , ' \vdash A_1'' = A :: \Omega \Rightarrow \, , \, , x{:}A_2; , ' \vdash A_1'' = A' :: \Omega$ (E-TRAN). The desired result then follows directly from an application of Lemma 10.4.4. (Precondition five of the lemma is met by the induction hypothesis since $\lfloor A_1'' \rfloor \leq \lfloor A_1 \rfloor$.)

□

The result extends immediately to the one-step subtyping relation:

Corollary 10.4.6 (Replacement with a one-step subtype)
If $, \vdash A_1 \lhd A_2$ *and* $, \, , x{:}A_1; , ' \vdash A \lhd A'$ *then*
$, \, , x{:}A_2; , ' \vdash A \lhd A'$.

Proof: By structural induction on the derivation of $, \, , x{:}A_1; , ' \vdash A \lhd A'$ using Theorem 10.4.5 as needed. □

Finally, using these results plus the replacement by an equal type result (Theorem 9.4.6), the desired replacement by a subtype result can be proved:

Theorem 10.4.7 (Replacement with a subtype)
If $, \vdash A_2 \leq A_1$ *then:*

1. *If* $\vdash \, , \, , x'{:}A_1; , '$ *valid then* $\vdash \, , \, , x'{:}A_2; , '$ *valid.*
2. *If* $, \, , x'{:}A_1; , ' \vdash A :: K$ *then* $, \, , x'{:}A_2; , ' \vdash A :: K$.
3. *If* $, \, , x'{:}A_1; , ' \vdash A = A' :: K$ *then* $, \, , x'{:}A_2; , ' \vdash A = A' :: K$.
4. *If* $, \, , x'{:}A_1; , ' \vdash x\rho \Rightarrow A$, $x \neq x'$, *then* $, \, , x'{:}A_2; , ' \vdash x\rho \Rightarrow A$.
5. *If* $, \, , x'{:}A_1; , ' \vdash A \leq A'$ *then* $, \, , x'{:}A_2; , ' \vdash A \leq A'$.

Proof: Proved simultaneously by structural induction on the derivations. The S-EQ case is handled by Theorem 9.4.6 and Lemma 10.2.5. The S-ONE case is handled by Theorem 10.4.5 and Corollary 10.4.6. □

10.5 A Semi-Decision Procedure

In this section I first prove some results about how subtyping behaves then give and prove correct a semi-decision procedure for subtyping. The first result I show for subtyping is that it acts in a component-wise manner:

Lemma 10.5.1 (Component-wise subtyping)

1. *If* $, \vdash \Pi x{:}A_1.\, A_1' \leq \Pi x{:}A_2.\, A_2'$, $x \notin$ *dom*$(,)$, *then* $, \vdash A_2 \leq A_1$ *and* $, , x{:}A_2 \vdash A_1' \leq A_2'$.

2. *If* $, \vdash \Sigma x{:}A_1.\, A_1' \leq \Sigma x{:}A_2.\, A_2'$, $x \notin$ *dom*$(,)$, *then* $, \vdash A_1 \leq A_2$ *and* $, , x{:}A_1 \vdash A_1' \leq A_2'$.

Proof: The proof of both parts are similar. I give the proof for only part one here: Inspection of the rules for subtyping shows that a subtyping derivation for $, \vdash \Pi x{:}A_1.\, A_1' \leq \Pi x{:}A_2.\, A_2'$ can be regarded as a linear sequence of types starting with $\Pi x{:}A_1.\, A_1'$ and ending with $\Pi x{:}A_2.\, A_2'$ such that adjacent types are related either by equality or one-step subtyping (under assignment $,$).

Using Lemma 10.3.7 we can turn one-step subtyping on types with shape ? into equalities on the same types. By using E-TRAN afterwards, we can collapse adjacent equalities so that all the remaining types in the sequence have shapes other than ?.

Hence, by Lemma 10.3.11 and the definition of intrinsic shapes, all the types in the new sequence have shape $\Pi \Rightarrow \exists n \geq 2.\ \forall i \in \{1 \ldots n\}.\ \exists B_i, B_i'$ such that

1. $A_1 = B_1$ and $A_1' = B_1'$

2. if $i > 1$ then either $, \vdash \Pi x{:}B_{i-1}.\, B_{i-1}' = \Pi x{:}B_i.\, B_i' :: \Omega$ or $, \vdash \Pi x{:}B_{i-1}.\, B_{i-1}' \lhd \Pi x{:}B_i.\, B_i'$

3. $A_2 = B_n$ and $A_2' = B_n'$

$\Rightarrow \forall i \in \{2 \ldots n\}.\ (, \vdash B_{i-1} = B_i :: \Omega$ or $, \vdash B_i \lhd B_{i-1})$ and $(, , x{:}B_i \vdash B_{i-1}' = B_i' :: \Omega$ or $, , x{:}B_i \vdash B_{i-1}' \lhd B_i')$ (Lemma 9.4.10 and O-DFUN)

$\Rightarrow \forall i \in \{1 \ldots n\}.\ , \vdash B_n \leq B_i$ (E-REFL, E-SYM, S-EQ, S-ONE, and S-TRAN applied repeatedly)

$\Rightarrow\ , \vdash A_2 \leq A_1$ and $\forall i \in \{2 \ldots n\}.\ , , x{:}B_n \vdash B_{i-1}' = B_i' :: \Omega$ or $, , x{:}B_n \vdash B_{i-1}' \lhd B_i'$ (Theorem 10.4.7)

$\Rightarrow\ , , x{:}A_2 \vdash A_1' \leq A_2'$ (E-REFL, S-EQ, S-ONE, and S-TRAN applied repeatedly) $\square$

With the next series of results, I establish that versions of the O-DFUN and O-DSUM rules using the full subtyping relation instead of the one-step subtyping relation are derivable:

Lemma 10.5.2 *Suppose* $, \vdash A_1 \leq A_2$. *Then:*

1. *If* $, \vdash \Pi x{:}A_2.\, A :: \Omega$ *then* $, \vdash \Pi x{:}A_2.\, A \leq \Pi x{:}A_1.\, A$.

2. *If* $, \vdash \Sigma x{:}A_2.\, A :: \Omega$ *then* $, \vdash \Sigma x{:}A_1.\, A \leq \Sigma x{:}A_2.\, A$.

Proof: The proof of both parts are similar. I give the proof for only part one here: WLOG, let $x \notin \mathrm{dom}(,)$. By C-DFUN, $, , x{:}A_2 \vdash A :: \Omega$. The proof proceeds by structural induction on the derivation of $, \vdash A_1 \leq A_2$:

O-EQ: Given $, \vdash A_1 = A_2 :: \Omega$. By E-REFL, $, , x{:}A_2 \vdash A = A :: \Omega$
$\Rightarrow\ , \vdash \Pi x{:}A_2.\, A = \Pi x{:}A_1.\, A :: \Omega$ (E-DFUN) $\Rightarrow\ , \vdash \Pi x{:}A_2.\, A \leq \Pi x{:}A_1.\, A$ (S-EQ).

O-ONE: Given $, \vdash A_1 \lhd A_2$. By O-REFL, $, , x{:}A_2 \vdash A \lhd A \Rightarrow\ , , x{:}A_1 \vdash A \lhd A$ (Corollary 10.4.6) $\Rightarrow\ , \vdash \Pi x{:}A_2.\, A \lhd \Pi x{:}A_1.\, A$ (O-DFUN)
$\Rightarrow\ , \vdash \Pi x{:}A_2.\, A \leq \Pi x{:}A_1.\, A$ (S-ONE).

O-TRAN: Given $, \vdash A_1 \leq A_0$ and $, \vdash A_0 \leq A_2$. By the inductive hypothesis,
$, \vdash \Pi x{:}A_2.\, A \leq \Pi x{:}A_0.\, A \Rightarrow\ , \vdash \Pi x{:}A_0.\, A :: \Omega$ (Lemma 10.1.2)
$\Rightarrow\ , \vdash \Pi x{:}A_0.\, A \leq \Pi x{:}A_1.\, A$ (induction hypothesis)
$\Rightarrow\ , \vdash \Pi x{:}A_2.\, A \leq \Pi x{:}A_1.\, A$ (E-TRAN).

□

Lemma 10.5.3 *If* $, , x{:}A \vdash A_1 \leq A_2$ *then:*

1. $, \vdash \Pi x{:}A.\, A_1 \leq \Pi x{:}A.\, A_2$

2. $, \vdash \Sigma x{:}A.\, A_1 \leq \Sigma x{:}A.\, A_2$

Proof: The proof of both parts are similar. I give the proof for only part one here: By Lemma 10.2.2, $, , x{:}A \vdash A_1 :: \Omega \Rightarrow\ , \vdash A :: \Omega$ (Lemma 6.3.5 and DECL-T). The proof proceeds by structural induction on the derivation of $, , x{:}A \vdash A_1 \leq A_2$:

O-EQ: Given $, , x{:}A \vdash A_1 = A_2 :: \Omega \Rightarrow\ , \vdash \Pi x{:}A.\, A_1 = \Pi x{:}A.\, A_2 :: \Omega$ (E-REFL and E-DFUN)
$\Rightarrow\ , \vdash \Pi x{:}A.\, A_1 \leq \Pi x{:}A.\, A_2$ (S-EQ).

O-ONE: Given $, , x{:}A \vdash A_1 \lhd A_2 \Rightarrow\ , \vdash \Pi x{:}A.\, A_1 \lhd \Pi x{:}A.\, A_2$ (O-REFL and O-DFUN) $\Rightarrow$
$, \vdash \Pi x{:}A.\, A_1 \leq \Pi x{:}A.\, A_2$ (S-ONE).

O-TRAN: Given $, , x{:}A \vdash A_1 \leq A_0$ and $, , x{:}A \vdash A_0 \leq A_2$. By the inductive hypothesis,
$, \vdash \Pi x{:}A.\, A_1 \leq \Pi x{:}A.\, A_0$ and $, \vdash \Pi x{:}A.\, A_0 \leq \Pi x{:}A.\, A_2$
$\Rightarrow\ , \vdash \Pi x{:}A.\, A_1 \leq \Pi x{:}A.\, A_2$ (S-TRAN).

□

Theorem 10.5.4

1. *If* $, \vdash A_2 \leq A_1$, $, , x{:}A_1 \vdash A_1' :: \Omega$, *and* $, , x{:}A_2 \vdash A_1' \leq A_2'$ *then* $, \vdash \Pi x{:}A_1.\, A_1' \leq \Pi x{:}A_2.\, A_2'$.

2. *If* $, \vdash A_1 \leq A_2$, $, , x{:}A_2 \vdash A_2' :: \Omega$, *and* $, , x{:}A_1 \vdash A_1' \leq A_2'$ *then* $, \vdash \Sigma x{:}A_1.\, A_1' \leq \Sigma x{:}A_2.\, A_2'$.

Proof: The proof of both parts are similar. I give the proof for only part one here: By Lemma 10.5.3, $, \vdash \Pi x{:}A_2.\, A_1' \leq \Pi x{:}A_2.\, A_2'$. By C-DFUN, $, \vdash \Pi x{:}A_1.\, A_1' :: \Omega \Rightarrow$ $, \vdash \Pi x{:}A_1.\, A_1' \leq \Pi x{:}A_2.\, A_1'$ (Lemma 10.5.2) $\Rightarrow$ $, \vdash \Pi x{:}A_1.\, A_1' \leq \Pi x{:}A_2.\, A_2'$ (S-TRAN). □

Using these results and those proved earlier in the chapter, I can reduce the problem of deciding subtyping on semi-canonical types to either simpler problems (e.g., equality or shape inspection) or subtyping problems involving sub-components of the original types:

Theorem 10.5.5 *Suppose* $A_1 \in C_{,}$ *and* $A_2 \in C_{,}$. *Then* $, \vdash A_1 \leq A_2$ *iff at least one of the following:*

1. $, \vdash A_1 = A_2 :: \Omega$

2. $\exists K.\ \lfloor A_1 \rfloor = \texttt{<=::}K\texttt{>}$ *and* $\lfloor A_2 \rfloor = \texttt{<}K\texttt{>}$

3. $\exists x, A_1', A_1'', A_2', A_2''.\ A_1 = \Pi x{:}A_1'.\, A_1''$, $A_2 = \Pi x{:}A_2'.\, A_2''$, $, \vdash A_2' \leq A_1'$, *and* $, , x{:}A_2' \vdash A_1'' \leq A_2''$.

4. $\exists x, A_1', A_1'', A_2', A_2''.\ A_1 = \Sigma x{:}A_1'.\, A_1''$, $A_2 = \Sigma x{:}A_2'.\, A_2''$, $, \vdash A_1' \leq A_2'$, *and* $, , x{:}A_1' \vdash A_1'' \leq A_2''$.

Proof:

$\Leftarrow$) Casing on which choice chosen:

case 1: By S-EQ, we have $, \vdash A_1 \leq A_2$.

case 2: By the definition of constructor shapes, $\exists A_1'$ such that $A_1 = \texttt{<=}A_1'\texttt{::}K\texttt{>}$ and $A_2 = \texttt{<}K\texttt{>}$. By Lemma 9.5.2, $, \vdash A_1 :: \Omega \Rightarrow$ $, \vdash A_1' :: K$ (C-TRANS) $\Rightarrow$ $, \vdash A_1 \lhd A_2$ (O-FORGET) $\Rightarrow$ $, \vdash A_1 \leq A_2$ (S-ONE).

case 3: By Lemma 9.5.2, $, \vdash A_1 :: \Omega$. By Theorem 10.5.4 then, $, \vdash A_1 \leq A_2$.

```
$SUT(, , A_1, A_2)$ = let $A'_1 = CAN(, , A_1)$;
                          $A'_2 = CAN(, , A_2)$ in
                      if $EQL(, , A'_1, A'_2, \Omega)$ then
                        return
                      else
                        case $(A'_1, A'_2)$ of
                      $(<=A''_1::K>, <K>)$: return
                    $(\Pi x{:}A_4.\ A_5, \Pi x{:}A_6.\ A_7)$: $SUT(, , A_6, A_4)$;
                                              $SUT((, , x{:}A_6), A_5, A_7)$;
                                              return
                    $(\Sigma x{:}A_4.\ A_5, \Sigma x{:}A_6.\ A_7)$: $SUT(, , A_4, A_6)$;
                                              $SUT((, , x{:}A_4), A_5, A_7)$;
                                              return
                                          $A$: raise fail
```

Figure 10.1: A semi-decision procedure for subtyping

case 4: By Lemma 9.5.2, , $\vdash A_2 :: \Omega$. By Theorem 10.5.4 then, , $\vdash A_1 \leq A_2$.

$\Rightarrow$) By Lemma 10.3.9, $\lceil A_1 \rceil_, = \lceil A_1 \rceil$ and $\lceil A_2 \rceil_, = \lceil A_2 \rceil$. By Corollary 10.3.12 then, $\lceil A_1 \rceil = \lceil A_2 \rceil$ or $\exists K.\ \lceil A_1 \rceil = <=::K>$ and $\lceil A_2 \rceil = <K>$. If the later is true, then we are done (case 2 holds) so assume the former. By Corollary 10.3.14 then, $\lceil A_2 \rceil_, \in \{\Pi, \Sigma\}$ or , $\vdash A_1 = A_2 :: \Omega$. If the later is true, then we are done (case 1 holds) so assume the former. By Lemma 10.5.1 then, case 3 or 4 holds.

□

This result can be used to construct a semi-decision procedure for subtyping:

Theorem 10.5.6 (Semi-decidability)
The judgment , $\vdash A_1 \leq A_2$ is semi-decidable. Non-termination only occurs if the judgment is false.

Proof: Using Theorems 10.5.5, 9.4.3, 9.4.11, and 9.5.10, a semi-recursive procedure is easily created. An example procedure using the notations of Section 8.3 is contained in Figure 10.1.

Here, $CAN(, , A)$ is an algorithm based on Theorem 9.5.10 that, if , $\vdash A :: \Omega$ holds, returns A' such that $A' \in C_,$ and , $\vdash A = A' :: \Omega$. Otherwise, it raises fail. $EQL(, , A_1, A_2, K)$, based on Theorem 9.4.11, returns true if $, \vdash A_1 = A_2 :: K$ holds and returns false otherwise.

Inspection plus the theorems mentioned previously suffice to show that:

1. $SUT(, , A_1, A_2)$ returns iff $, \vdash A_1 \leq A_2$.
2. $SUT(, , A_1, A_2)$ fails $\Rightarrow \neg, \vdash A_1 \leq A_2$.

$\Box$

10.6 The Simple Type System

In this and the next three sections, I show that the subtyping relation for my system is undecidable using a slight modification to Benjamin Pierce's proof of the undecidability of $F_{\leq}$ subtyping [51, 50]. In order to make the source of the undecidability clear, I shall first introduce the *simple* type system, which has only the minimum constructs and rules required to produce undecidability; second show that the simple subtyping problem is easier than the kernel system's subtyping problem; and third show that the simple subtyping problem is undecidable.

The simple system has only types; there are no constructors of kind other than Ω. The methods for building types allow only for binary opaque weak sums ($\exists\alpha.A$), binary transparent weak sums ($\exists\alpha{=}A_1.A_2$), and *negative* types ($\neg A$). Negative types abstract the idea of contra-variant subtyping: $, \vdash \neg A \leq \neg A'$ iff $, \vdash A' \leq A$. In a more realistic type system, contra-variance would be part of a more complicated construct (e.g., Π-types). The syntax for the simple type system is as follows:

Definition [Simple] 10.6.1 (Syntax)

Types	A	::=	$\alpha \mid \neg A \mid \exists\alpha.A \mid \exists\alpha{=}A_1.A_2$
Assignments	$,$	::=	$\bullet \mid , , \alpha$

Here, the metavariable α ranges over *type* variables. Equivalence of types, assignments, and other syntactic objects is defined as usual (α-conversion modulo the assignment exception[2]), and the definitions of free type variables (FV(A)), type substitution ($[A/\alpha]A'$) , and the domain function (dom(,)) are the obvious ones:

[2]Namely, assignments with no redeclared variables are never considered equivalent to assignments with redeclared variables.

Definition [Simple] 10.6.2 (Free type variables)

$$\begin{array}{rcl}
FV(\alpha) & = & \{\alpha\} \\
FV(\neg A) & = & FV(A) \\
FV(\exists\alpha.A) & = & FV(A) \Leftrightarrow \{\alpha\} \\
FV(\exists\alpha{=}A_1.A_2) & = & FV(A_1) \cup (FV(A_2) \Leftrightarrow \{\alpha\})
\end{array}$$

Definition [Simple] 10.6.3 (Type substitution)

$$\begin{array}{rcll}
[A/\alpha]\alpha & = & A & \\
[A/\alpha]\alpha' & = & \alpha' & (\alpha \neq \alpha') \\
[A/\alpha](\neg A_1) & = & \neg[A/\alpha]A_1 & \\
[A/\alpha]\exists\alpha'.A_1 & = & \exists\alpha'.[A/\alpha]A_1 & (\alpha' \neq \alpha,\ \alpha' \notin FV(A)) \\
[A/\alpha]\exists\alpha'{=}A_1.A_2 & = & \exists\alpha'{=}[A/\alpha]A_1.[A/\alpha]A_2 & (\alpha' \neq \alpha,\ \alpha' \notin FV(A))
\end{array}$$

Definition [Simple] 10.6.4 (Domain function)

$$\begin{array}{rcl}
dom(\bullet) & = & \emptyset \\
dom(,\ ,\alpha) & = & dom(,\ ) \cup \{\alpha\}
\end{array}$$

The simple type system has the following judgments:

Definition [Simple] 10.6.5 (Judgments)

$$\begin{array}{rl}
\vdash ,\ \ valid & \textit{valid assignment} \\
,\ \vdash A\ valid & \textit{valid type} \\
,\ \vdash A = A' & \textit{equal types} \\
,\ \vdash A \leq A' & \textit{subtyping relation}
\end{array}$$

The rules for these judgments follow. In order to mirror Pierce's fragment $F_{\leq}^{\mathrm{F}}$ (see Section 6.4 of Pierce's thesis [51]) closely, these rules handle transparent definitions $(\alpha{=}A)$ by substitution rather than by adding a transparent definition to the current assignment. (See, for example, the SUM-T and SU-SUM-T rules.)

Definition [Simple] 10.6.6 (Assignment Formation Rules)

$$\vdash \bullet\ valid \qquad \text{(EMP)}$$

$$\frac{\vdash ,\ \ valid \qquad \alpha \notin dom(,\ )}{\vdash ,\ ,\alpha\ valid} \qquad \text{(DCL)}$$

Definition [Simple] 10.6.7 (Type Formation Rules)

$$\frac{\vdash ,\ \ \mathit{valid} \qquad \alpha \in \mathit{dom}(,\ )}{,\ \vdash \alpha\ \mathit{valid}} \qquad \text{(VAR)}$$

$$\frac{,\ \vdash A\ \mathit{valid}}{,\ \vdash \neg A\ \mathit{valid}} \qquad \text{(NEG)}$$

$$\frac{,\ , \alpha \vdash A\ \mathit{valid}}{,\ \vdash \exists\alpha.A\ \mathit{valid}} \qquad \text{(SUM-O)}$$

$$\frac{,\ \vdash A\ \mathit{valid} \qquad ,\ \vdash [A/\alpha]A'\ \mathit{valid}}{,\ \vdash \exists\alpha{=}A.A'\ \mathit{valid}} \qquad \text{(SUM-T)}$$

Definition [Simple] 10.6.8 (Equality Rules)

$$\frac{,\ \vdash \alpha\ \mathit{valid}}{,\ \vdash \alpha = \alpha} \qquad \text{(EQ-VAR)}$$

$$\frac{,\ \vdash A_1 = A_2}{,\ \vdash \neg A_1 = \neg A_2} \qquad \text{(EQ-NEG)}$$

$$\frac{,\ , \alpha \vdash A = A'}{,\ \vdash \exists\alpha.A = \exists\alpha.A'} \qquad \text{(EQ-SUM-O)}$$

$$\frac{,\ \vdash A_1 = A_1' \qquad ,\ \vdash [A_1/\alpha]A_2 = [A_1/\alpha]A_2'}{,\ \vdash \exists\alpha{=}A_1.A_2 = \exists\alpha{=}A_1'.A_2'} \qquad \text{(EQ-SUM-T)}$$

Definition [Simple] 10.6.9 (Subtyping Rules)

$$\frac{,\ \vdash \alpha\ \mathit{valid}}{,\ \vdash \alpha \leq \alpha} \qquad \text{(SU-VAR)}$$

$$\frac{,\ \vdash A_2 \leq A_1}{,\ \vdash \neg A_1 \leq \neg A_2} \qquad \text{(SU-NEG)}$$

$$\frac{,\ , \alpha \vdash A \leq A'}{,\ \vdash \exists\alpha.A \leq \exists\alpha.A'} \qquad \text{(SU-SUM-O)}$$

$$\frac{,\ \vdash A_1 = A_1' \qquad ,\ \vdash [A_1/\alpha]A_2 \leq [A_1/\alpha]A_2'}{,\ \vdash \exists\alpha{=}A_1.A_2 \leq \exists\alpha{=}A_1'.A_2'} \qquad \text{(SU-SUM-T)}$$

$$\frac{\begin{array}{c}, \vdash A \textit{ valid} \\ , , \alpha \vdash A_2 \textit{ valid} \qquad , \vdash [A/\alpha]A_1 \leq [A/\alpha]A_2\end{array}}{, \vdash \exists\alpha{=}A.A_1 \leq \exists\alpha.A_2} \qquad \text{(SU-FOR)}$$

Note that all the rules are syntax directed: each shape or pair of shapes has at most one applicable rule under each relation. Because of this fact, decision procedures can be constructed directly from the rules. The termination of such procedures is unclear, however, because the SUM-T, EQ-SUM-T, SU-SUM-T, and SU-FOR rules may increase the size of the types being dealt with.

The usual propositions on judgments hold:

Lemma [Simple] 10.6.10 (Structural property I)

1. *If* $\vdash , , \alpha$ *valid then* $\vdash ,$ *valid.*
2. *If* $, \vdash A$ *valid then* $\vdash ,$ *valid.*

Lemma [Simple] 10.6.11 (Weakening)
Suppose $\vdash , ; ,'$ *valid and* $dom(,') \cap dom(,'') = \emptyset$*. Then:*

1. *If* $\vdash , ; ,''$ *valid then* $\vdash , ; ,'; ,''$ *valid.*
2. *If* $, ; ,'' \vdash A$ *valid then* $, ; ,'; ,'' \vdash A$ *valid.*

Proof: By sequential structural induction on the length of the derivations involving $, ; ,''$. □

Although the simple-subtyping relation lacks an EQ rule, the next lemma shows that the following SU-EQ rule is derivable:

$$\frac{, \vdash A_1 = A_2}{, \vdash A_1 \leq A_2} \qquad \text{(SU-EQ)}$$

Lemma [Simple] 10.6.12 *If* $, \vdash A_1 = A_2$ *then* $, \vdash A_1 \leq A_2$ *and* $, \vdash A_2 \leq A_1$.

Proof: Proved by structural induction on the derivation. □

The following notion of an *extended* simple-subtyping derivation that permits the use of SU-EQ will be useful later:

Definition [Simple] 10.6.13 (Extended derivations)
An extended derivation for $, \vdash A_1 \leq A_2$ *is a derivation of* $, \vdash A_1 \leq A_2$ *using the normal simple-subtyping rules plus the SU-EQ rule. The* size *of an extended derivation is defined to be the number of SU rules (including SU-EQ) used in the derivation.*

Because of Lemma 10.6.12, an extended derivation for $, \vdash A_1 \leq A_2$ suffices to prove that $, \vdash A_1 \leq A_2$. Extended derivations will be important because their size as defined above remains constant under operations such as strengthening and type substitution:

Lemma [Simple] 10.6.14 (Strengthening)

1. *If* $\vdash , , \alpha; ,'$ *valid then* $\vdash , ; ,'$ *valid.*
2. *If* $, , \alpha; ,' \vdash A$ *valid and* $\alpha \notin FV(A)$ *then* $, ; ,' \vdash A$ *valid.*
3. *If* $, , \alpha; ,' \vdash A_1 = A_2$ *and* $\alpha \notin FV(A_1) \cup FV(A_2)$ *then* $, ; ,' \vdash A_1 = A_2$.
4. *If* $, , \alpha; ,' \vdash A_1 \leq A_2$ *by an extended derivation and* $\alpha \notin FV(A_1) \cup FV(A_2)$ *then* $, ; ,' \vdash A_1 \leq A_2$ *by an extended derivation of equal size.*

Proof: Proved sequentially by structural induction on the derivations. □

Lemma [Simple] 10.6.15 *Suppose* $, \vdash A$ *valid. Then:*

1. *If* $, , \alpha; ,' \vdash A_1$ *valid then* $, , \alpha; ,' \vdash [A/\alpha]A_1$ *valid.*
2. *If* $, , \alpha; ,' \vdash A_1 = A_2$ *then* $, , \alpha; ,' \vdash [A/\alpha]A_1 = [A/\alpha]A_2$.
3. *If* $, , \alpha; ,' \vdash A_1 \leq A_2$ *by an extended derivation then*
   $, , \alpha; ,' \vdash [A/\alpha]A_1 \leq [A/\alpha]A_2$ *by an extended derivation of equal size.*

Proof: Proved sequentially by structural induction on the derivations involving A_1. Lemma 10.6.11 and Lemma 10.6.10 are used to handle the VAR case. The SU-EQ rule is used to handle the SU-VAR case in order to keep the resulting derivation the same size. □

Corollary [Simple] 10.6.16 (Validity of type substitution)
Suppose $, \vdash A$ *valid. Then:*

1. *If* $, , \alpha \vdash A_1 = A_2$ *then* $, \vdash [A/\alpha]A_1 = [A/\alpha]A_2$.
2. *If* $, , \alpha \vdash A_1 \leq A_2$ *by an extended derivation then*
   $, \vdash [A/\alpha]A_1 \leq [A/\alpha]A_2$ *by an extended derivation of equal size.*

Proof: Follows immediately from Lemma 10.6.15 and Theorem 10.6.14. □

10.7 Encoding Simple Judgments

I shall show that the simple subtyping problem is easier than than the kernel system's subtyping problem by showing that simple subtyping problems can be translated into equivalent kernel-system subtyping problems using the following encoding on simple-system types, assignments, and judgments. The encoding results in kernel-system constructors, assignments, and judgments respectively.

Definition [Simple] 10.7.1 (Encoding simple judgments)

$$\begin{array}{lcl}
\alpha^{\otimes} & = & \widehat{\alpha}! \\
(\neg A)^{\otimes} & = & \Pi x{:}A^{\otimes}.\,{<}\Omega{>} \\
(\exists\alpha.A_1)^{\otimes} & = & \Sigma\widehat{\alpha}{:}{<}\Omega{>}.\,A_1^{\otimes} \\
(\exists\alpha{=}A_1.A_2)^{\otimes} & = & \Sigma\widehat{\alpha}{:}{<}{=}A_1^{\otimes}{::}\Omega{>}.\,A_2^{\otimes} \\
 & & \\
\bullet^{\otimes} & = & \bullet \\
(,\ ,\alpha)^{\otimes} & = & ,\ ^{\otimes},\widehat{\alpha}{:}{<}\Omega{>} \\
 & & \\
(\vdash\ ,\ \ \mathit{valid})^{\otimes} & = & \vdash\ ,\ ^{\otimes}\ \mathit{valid} \\
(,\ \vdash A\ \mathit{valid})^{\otimes} & = & ,\ ^{\otimes}\vdash A^{\otimes} :: \Omega \\
(,\ \vdash A_1 = A_2)^{\otimes} & = & ,\ ^{\otimes}\vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega \\
(,\ \vdash A_1 \leq A_2)^{\otimes} & = & ,\ ^{\otimes}\vdash A_1^{\otimes} \leq A_2^{\otimes}
\end{array}$$

Here, $\widehat{\Leftrightarrow}$ is any bijective mapping from simple type variables to kernel-system term variables.

Encoding type substitution in the simple system results in the substitution in the kernel system of a constructor for a constructor extraction:

Definition 10.7.2 (Constructor substitution for an extraction)

$$
\begin{array}{rcll}
[A/x\rho!]x\rho! & = & A & \\
[A/x\rho!]x'\rho'! & = & x'\rho'! & (x\rho \neq x'\rho') \\
[A/x\rho!]\alpha & = & \alpha & \\
[A/x\rho!]\lambda\alpha'{::}K.\,A' & = & \lambda\alpha'{::}K.\,[A/x\rho!]A' & (\alpha' \notin FCV(A)) \\
[A/x\rho!]\Pi x'{:}A_1.\,A_2 & = & \Pi x'{:}[A/x\rho!]A_1.\,[A/x\rho!]A_2 & (x' \neq x,\ x' \notin FTV(A)) \\
[A/x\rho!]\Sigma x'{:}A_1.\,A_2 & = & \Sigma x'{:}[A/x\rho!]A_1.\,[A/x\rho!]A_2 & (x' \neq x,\ x' \notin FTV(A)) \\
[A/x\rho!](A_1\,A_2) & = & [A/x\rho!]A_1\,[A/x\rho!]A_2 & \\
[A/x\rho!]{<=}A'{::}K{>} & = & {<=}[A/x\rho!]A'{::}K{>} & \\
[A/x\rho!]{<}K{>} & = & {<}K{>} & \\
[A/x\rho!]\mathbf{rec} & = & \mathbf{rec} & \\
[A/x\rho!]\mathbf{ref} & = & \mathbf{ref} & \\
 & & & \\
[A/x\rho!](\alpha{::}K) & = & \alpha{::}K & \\
[A/x\rho!](x'{:}A') & = & x'{:}[A/x\rho!]A' & \\
 & & & \\
[A/x\rho!]\bullet & = & \bullet & \\
[A/x\rho!](,\ , D) & = & ([A/x\rho!],\ ), [A/x\rho!]D &
\end{array}
$$

Lemma [Simple] 10.7.3 (Encoding properties)

1. $dom(,\ ^{\otimes}) = \widehat{dom(,\ )}$
2. $([A_1/\alpha]A_2)^{\otimes} = [A_1^{\otimes}/\hat{\alpha}!](A_2^{\otimes})$

Proof: Proved using induction on , and A_2 respectively. □

The intrinsic shapes of encoded simple types can be computed using previous results. Note that encoded variables have intrinsic shape ? because simple assignments do not have transparent definitions.

Lemma [Simple] 10.7.4 (Shapes of encoded types)

1. *If* $,\ ^{\otimes} \vdash \alpha^{\otimes} :: \Omega$ *then* $\lceil \alpha^{\otimes} \rceil_{,\ ^{\otimes}} = ?$.
2. *If* $,\ ^{\otimes} \vdash (\neg A')^{\otimes} :: \Omega$ *then* $\lceil (\neg A')^{\otimes} \rceil_{,\ ^{\otimes}} = \Pi$.
3. *If* $,\ ^{\otimes} \vdash (\exists\alpha.A_1)^{\otimes} :: \Omega$ *then* $\lceil (\exists\alpha.A_1)^{\otimes} \rceil_{,\ ^{\otimes}} = \Sigma$.

4. If $,^{\otimes} \vdash (\exists\alpha{=}A_1.A_2)^{\otimes} :: \Omega$ *then* $\lceil(\exists\alpha{=}A_1.A_2)^{\otimes}\rceil_{,\otimes} = \Sigma$.

Proof: For part one, we have $,^{\otimes} \vdash \alpha^{\otimes} :: \Omega \Rightarrow ,^{\otimes} \vdash \hat{\alpha}! :: \Omega \Rightarrow \hat{\alpha} \in \mathrm{dom}(,^{\otimes})$ (Theorem 6.5.6) $\Rightarrow \alpha \in \mathrm{dom}(,)$ (Lemma 10.7.3) $\Rightarrow \hat{\alpha}{:}{<}\Omega{>} \in ,^{\otimes} \Rightarrow ,^{\otimes}(\hat{\alpha}) = {<}\Omega{>}$ (Lemma 6.3.5 and Lemma 9.4.2) $\Rightarrow \hat{\alpha}! \in C_{,\otimes}$. (Lemma 9.5.11) $\Rightarrow \lceil\hat{\alpha}!\rceil_{,\otimes} = \lceil\hat{\alpha}!\rceil = ?$ (Lemma 10.3.9).

Parts two to four follow immediately from the definitions of the encoding, intrinsic shape, and E-REFL. □

In order to show that simple subtyping problems can be translated into equivalent kernel-system subtyping problems, I shall need the following series of lemmas relating substitutions for constructor extractions and transparent definitions ($x{:}{<}{=}A{::}\Omega{>}$) in the kernel system:

Lemma 10.7.5 *If* $,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]A' :: K$ *then*
$,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]A' = A' :: K$.

Proof: Proved by structural induction on A'. Interesting cases:

DFUN: Here $,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]\Pi x'{:}A_1.\ A_2 :: \Omega,\ x' \notin \mathrm{dom}(,\ ,x{:}{<}{=}A{::}\Omega{>};,')$
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash \Pi x'{:}[A/x!]A_1.\ [A/x!]A_2 :: \Omega$
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,',x'{:}[A/x!]A_1 \vdash [A/x!]A_2 :: \Omega$ (C-DFUN)
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]A_1 :: \Omega$ (Lemma 6.3.5)
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]A_1 = A_1 :: \Omega$ and
$,\ ,x{:}{<}{=}A{::}\Omega{>};,',x'{:}[A/x!]A_1 \vdash [A/x!]A_2 = A_2 :: \Omega$ (induction hypothesis)
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash \Pi x'{:}[A/x!]A_1.\ [A/x!]A_2 = \Pi x'{:}A_1.\ A_2 :: \Omega$ (E-DFUN)
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]\Pi x'{:}A_1.\ A_2 = \Pi x'{:}A_1.\ A_2 :: K$

EXT I: Here $,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]x! :: K \Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash A :: K$. By Lemma 6.3.5, $\vdash ,\ ,x{:}{<}{=}A{::}\Omega{>};,'$ valid and $\vdash ,\ ,x{:}{<}{=}A{::}\Omega{>}$ valid $\Rightarrow ,\ \vdash {<}{=}A{::}\Omega{>} :: \Omega$ (DECL-T) $\Rightarrow ,\ \vdash A :: \Omega$ (C-TRANS) $\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash A :: \Omega$ (Theorem 9.4.1) $\Rightarrow K = \Omega$ (Theorem 9.4.5). By P-INIT, $,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash x \Rightarrow {<}{=}A{::}\Omega{>}$
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash x! = A :: \Omega$ (C-EXT-T, E-ABBREV, E-REFL, and C-TRANS)
$\Rightarrow ,\ ,x{:}{<}{=}A{::}\Omega{>};,' \vdash [A/x!]x! = x! :: \Omega$ (E-SYM)

□

Lemma 10.7.6 *If* $,\ \vdash A :: \Omega$ *and* $,\ \vdash [A/x!]A' :: \Omega$, $x \notin$ *dom*$(,)$, *then*
$,\ ,x{:}{<}{=}A{::}\Omega{>} \vdash [A/x!]A' = A' :: \Omega$.

Proof: By C-TRANS, $,\ \vdash\ <=A::\Omega> :: \Omega$. By Lemma 6.3.5, $\vdash\ ,$ valid $\Rightarrow\ \vdash\ ,\ , x{:}<=A::\Omega>$ valid (DECL-T) $\Rightarrow\ ,\ , x{:}<=A::\Omega> \vdash [A/x!]A' :: \Omega$ (Theorem 9.4.1) $\Rightarrow\ ,\ , x{:}<=A::\Omega> \vdash [A/x!]A' = A' :: \Omega$. (Lemma 10.7.5) □

Lemma 10.7.7 *If* $,\ , x{:}<=A::\Omega>;\ ,' \vdash A' :: K$ *then* $,\ , x{:}<=A::\Omega>;\ ,' \vdash A' = [A/x!]A' :: K$.

Proof: Proved by structural induction on A'. Interesting cases:

DFUN: Here $,\ , x{:}<=A::\Omega>;\ ,' \vdash \Pi x'{:}A_1.\, A_2 :: \Omega,\ x' \notin \mathrm{dom}(,\ , x{:}<=A::\Omega>;\ ,')$
$\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,', x'{:}A_1 \vdash A_2 :: \Omega$ (C-DFUN) $\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash A_1 :: \Omega$ (Lemma 6.3.5) $\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash A_1 = [A/x!]A_1 :: \Omega$ and $,\ , x{:}<=A::\Omega>;\ ,', x'{:}A_1 \vdash A_2 = [A/x!]A_2 :: \Omega$ (induction hypothesis)
$\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash \Pi x'{:}A_1.\, A_2 = \Pi x'{:}[A/x!]A_1.\, [A/x!]A_2 :: \Omega$ (E-DFUN)
$\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash \Pi x'{:}A_1.\, A_2 = [A/x!]\Pi x'{:}A_1.\, A_2 :: \Omega$

EXT I: Here $,\ , x{:}<=A::\Omega>;\ ,' \vdash x! :: K\ \Rightarrow\ \vdash\ ,\ , x{:}<=A::\Omega>;\ ,'$ valid (Lemma 6.3.5) $\Rightarrow$ $,\ , x{:}<=A::\Omega>;\ ,' \vdash x \Rightarrow\ <=A::\Omega>$ (P-INIT)
$\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash x! = A :: \Omega$ (C-EXT-T, E-ABBREV, E-REFL, and Theorem 9.4.4) $\Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash x! = [A/x!]x! :: \Omega$ By Theorem 9.4.5 and Theorem 9.4.3, $K = \Omega\ \Rightarrow\ ,\ , x{:}<=A::\Omega>;\ ,' \vdash x! = [A/x!]x! :: K$

□

Lemma [Simple] 10.7.8 *Suppose* $,\ , x{:}<=A::\Omega>;\ ,' \vdash A_1 \lhd A_2$ *by a derivation that uses s applications of the one-step subtyping rules. Then* $,\ , x{:}<=A::\Omega>;\ ,' \vdash [A/x!]A_1 \lhd [A/x!]A_2$ *by s applications of the one-step subtyping rules.*

Proof: Proved by structural induction on the derivation. Lemmas 10.1.2, 10.7.7, and Theorem 9.4.3 are used to establish that $[A/x!]A_1$ and $[A/x!]A_2$ are valid under $,\ , x{:}<=A::\Omega>;\ ,'$. Theorem 10.1.6 is needed in addition to handle the O-DFUN and O-DSUM cases; inspection of its proof shows that it leaves the number of applications of the one-step subtyping rules unchanged. □

The previous lemmas when combined with the following lemma, the structural properties, strengthening, weakening, and transitivity, allow converting between the use of transparent definitions and substitutions. For example, given $,\ \vdash A^{\otimes} :: \Omega$ and $x \notin \mathrm{dom}(,\ )$, we can show that $,\ , x{:}<=A^{\otimes}::\Omega> \vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega$ iff $,\ \vdash [A^{\otimes}/x!](A_1^{\otimes}) = [A^{\otimes}/x!](A_2^{\otimes}) :: \Omega$.

Lemma [Simple] 10.7.9 *If* $\hat{\alpha} \notin FTV(A'^{\otimes})$ *then* $\hat{\alpha} \notin FTV([A'^{\otimes}/\hat{\alpha}!](A^{\otimes}))$

10.8 Problem Reduction

In this section I show that for any simple judgment J, J holds in the simple system iff $J^{\otimes}$ holds in the kernel system. The forward direction is straightforward:

Lemma [Simple] 10.8.1 (Reduction I)

1. *If* $\vdash$, *valid then* $\vdash$, $^{\otimes}$ *valid.*
2. *If* , $\vdash A$ *valid then* , $^{\otimes} \vdash A^{\otimes} :: \Omega$.
3. *If* , $\vdash A_1 = A_2$ *then* , $^{\otimes} \vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega$.
4. *If* , $\vdash A_1 \leq A_2$ *then* , $^{\otimes} \vdash A_1^{\otimes} \leq A_2^{\otimes}$.

Proof: Proved by simultaneous induction on the derivations. Interesting cases:

DCL: Here $\vdash$, , α valid derived via rule DCL from $\vdash$, valid and $\alpha \notin$ dom(,) $\Rightarrow \widehat{\alpha} \notin$ dom(, $^{\otimes}$) (Lemma 10.7.3). By the induction hypothesis, $\vdash$, $^{\otimes}$ valid $\Rightarrow$, $^{\otimes} \vdash \texttt{<}\Omega\texttt{>} :: \Omega$ (C-OPAQ) $\Rightarrow \vdash$, $^{\otimes}, \widehat{\alpha}{:}\texttt{<}\Omega\texttt{>}$ valid (DECL-T) $\Rightarrow \vdash$ (, $\alpha)^{\otimes}$ valid.

VAR: Here, , $\vdash \alpha$ valid derived via VAR from $\vdash$, valid and $\alpha \in$ dom(,) $\Rightarrow \widehat{\alpha}{:}\texttt{<}\Omega\texttt{>} \in$, $^{\otimes}$ By the induction hypothesis, $\vdash$, $^{\otimes}$ valid $\Rightarrow$, $^{\otimes} \vdash \widehat{\alpha} \Rightarrow \texttt{<}\Omega\texttt{>}$ (P-INIT) $\Rightarrow$, $^{\otimes} \vdash \widehat{\alpha}! :: \Omega$ (C-EXT-O) $\Rightarrow$, $^{\otimes} \vdash \alpha^{\otimes} :: \Omega$.

SUM-T: Here, , $\vdash \exists\alpha{=}A.A'$ valid derived via SUM-T from , $\vdash A$ valid and , $\vdash [A/\alpha]A'$ valid, $\alpha \notin$ dom(,) $\Rightarrow \widehat{\alpha} \notin$ dom(, $^{\otimes}$) (Lemma 10.7.3). By applying the induction hypothesis, , $^{\otimes} \vdash A^{\otimes} :: \Omega$ and , $^{\otimes} \vdash ([A/\alpha]A')^{\otimes} :: \Omega \Rightarrow$, $^{\otimes} \vdash [A^{\otimes}/\widehat{\alpha}!](A'^{\otimes}) :: \Omega$ (Lemma 10.7.3)
$\Rightarrow$, $^{\otimes}, \widehat{\alpha}{:}\texttt{<=}A^{\otimes}{::}\Omega\texttt{>} \vdash [A^{\otimes}/\widehat{\alpha}!](A'^{\otimes}) = A'^{\otimes} :: \Omega$ (Lemma 10.7.6)
$\Rightarrow$, $^{\otimes}, \widehat{\alpha}{:}\texttt{<=}A^{\otimes}{::}\Omega\texttt{>} \vdash A'^{\otimes} :: \Omega$ (Theorem 9.4.3)
$\Rightarrow$, $^{\otimes} \vdash \Sigma\widehat{\alpha}{:}\texttt{<=}A^{\otimes}{::}\Omega\texttt{>}.\,A'^{\otimes} :: \Omega$ (C-DSUM) $\Rightarrow$, $^{\otimes} \vdash (\exists\alpha{=}A.A')^{\otimes} :: \Omega$.

EQ-SUM-T: Here, , $\vdash \exists\alpha{=}A_1.A_2 = \exists\alpha{=}A_1'.A_2'$, $\alpha \notin$ dom(,), derived via EQ-SUM-T from , $\vdash A_1 = A_1'$ and , $\vdash [A_1/\alpha]A_2 = [A_1/\alpha]A_2' \Rightarrow \widehat{\alpha} \notin$ dom(, $^{\otimes}$) (Lemma 10.7.3). By the induction hypothesis, , $^{\otimes} \vdash A_1^{\otimes} = A_1'^{\otimes} :: \Omega$ and
, $^{\otimes} \vdash ([A_1/\alpha]A_2)^{\otimes} = ([A_1/\alpha]A_2')^{\otimes} :: \Omega$
$\Rightarrow$, $^{\otimes} \vdash [A_1^{\otimes}/\widehat{\alpha}!](A_2^{\otimes}) = [A_1^{\otimes}/\widehat{\alpha}!](A_2'^{\otimes}) :: \Omega$ (Lemma 10.7.3)
$\Rightarrow$, $^{\otimes}, \widehat{\alpha}{:}\texttt{<=}A_1^{\otimes}{::}\Omega\texttt{>} \vdash [A_1^{\otimes}/\widehat{\alpha}!](A_2^{\otimes}) = A_2^{\otimes} :: \Omega$ and
, $^{\otimes}, \widehat{\alpha}{:}\texttt{<=}A_1^{\otimes}{::}\Omega\texttt{>} \vdash [A_1^{\otimes}/\widehat{\alpha}!](A_2'^{\otimes}) = A_2'^{\otimes} :: \Omega$ (Lemmas 9.4.3 and 10.7.6)
$\Rightarrow$, $^{\otimes}, \widehat{\alpha}{:}\texttt{<=}A_1^{\otimes}{::}\Omega\texttt{>} \vdash A_2^{\otimes} = A_2'^{\otimes} :: \Omega$ (E-SYM and E-TRAN)
$\Rightarrow$, $^{\otimes} \vdash \Sigma\widehat{\alpha}{:}\texttt{<=}A_1^{\otimes}{::}\Omega\texttt{>}.\,A_2^{\otimes} = \Sigma\widehat{\alpha}{:}\texttt{<=}A_1'^{\otimes}{::}\Omega\texttt{>}.\,A_2'^{\otimes} :: \Omega$ (E-TRANS and E-DSUM)
$\Rightarrow$, $^{\otimes} \vdash (\exists\alpha{=}A_1.A_2)^{\otimes} = (\exists\alpha{=}A_1'.A_2')^{\otimes} :: \Omega$

SU-FOR: Here, $, \vdash \exists\alpha{=}A.A_1 \leq \exists\alpha.A_2$, $\alpha \notin \mathrm{dom}(,\ )$, derived via SU-FOR from $, \vdash A$ valid, $, , \alpha \vdash A_2$ valid, and $, \vdash [A/\alpha]A_1 \leq [A/\alpha]A_2 \Rightarrow \widehat{\alpha} \notin \mathrm{dom}(, ^{\otimes})$ (Lemma 10.7.3). By the induction hypothesis, $, ^{\otimes} \vdash A^{\otimes} :: \Omega$, $, ^{\otimes}, \widehat{\alpha}{:}{<}\Omega{>} \vdash A_2^{\otimes} :: \Omega$, and
$, ^{\otimes} \vdash ([A/\alpha]A_1)^{\otimes} \leq ([A/\alpha]A_2)^{\otimes} \Rightarrow , ^{\otimes} \vdash [A^{\otimes}/\widehat{\alpha}!](A_1^{\otimes}) \leq [A^{\otimes}/\widehat{\alpha}!](A_2^{\otimes})$ (Lemma 10.7.3)
$\Rightarrow , ^{\otimes}, \widehat{\alpha}{:}{<}{=}A^{\otimes}{::}\Omega{>} \vdash [A^{\otimes}/\widehat{\alpha}!](A_1^{\otimes}) = A_1^{\otimes} :: \Omega$ and
$, ^{\otimes}, \widehat{\alpha}{:}{<}{=}A^{\otimes}{::}\Omega{>} \vdash [A^{\otimes}/\widehat{\alpha}!](A_2^{\otimes}) = A_2^{\otimes} :: \Omega$ (Lemmas 10.2.2 and 10.7.6)
$\Rightarrow , ^{\otimes}, \widehat{\alpha}{:}{<}{=}A_1^{\otimes}{::}\Omega{>} \vdash A_1^{\otimes} \leq A_2^{\otimes}$ (E-SYM, S-EQ, and S-TRAN).

By Lemma 6.3.5, DECL-T, C-TRANS, O-FORGET, S-ONE,
$, ^{\otimes} \vdash {<}{=}A_1^{\otimes}{::}\Omega{>} \leq {<}\Omega{>} \Rightarrow , ^{\otimes} \vdash \Sigma\widehat{\alpha}{:}{<}{=}A^{\otimes}{::}\Omega{>}.\ A_1^{\otimes} \leq \Sigma\widehat{\alpha}{:}{<}\Omega{>}.\ A_2^{\otimes}$
(Theorem 10.5.4) $\Rightarrow , ^{\otimes} \vdash (\exists\alpha{=}A.A_1)^{\otimes} \leq (\exists\alpha.A_2)^{\otimes}$

□

The backward direction is harder. To handle the simple validity and equality judgments, the following metric on types is useful:

Definition [Simple] 10.8.2 (Size metric)

$$\begin{array}{rcl} |\alpha| & = & 0 \\ |\neg A| & = & 1 + |A| \\ |\exists\alpha.A| & = & 1 + |A| \\ |\exists\alpha{=}A.A'| & = & 1 + |A| + |[A/\alpha]A'| \end{array}$$

This definition can be seen to be well defined by noticing that if we extend types so they may contain integers and define $|i| = i$, we get that $|[A_1/\alpha]A_2| = |[|A_1|/\alpha]A_2|$. Note that under this metric, both A_1 and $[A_1/\alpha]A_2$ are smaller than $\exists\alpha{=}A_1.A_2$. This fact will allow us to recurse on the "natural sub-components" of a simple type.

Using this metric proving the backward direction for the validity judgments is straightforward:

Lemma [Simple] 10.8.3 (Reduction II: validity)

1. *If* $\vdash , ^{\otimes}$ *valid then* $\vdash ,$ *valid.*

2. *If* $, ^{\otimes} \vdash A^{\otimes} :: \Omega$ *then* $, \vdash A$ *valid.*

Proof: Proved sequentially by induction on $|,\ |$ and $|A|$ respectively. Interesting cases:

Dcl: Here $\vdash (, , \alpha)^{\otimes}$ valid $\Rightarrow \vdash , ^{\otimes}, \widehat{\alpha}{:}{<}\Omega{>}$ valid $\Rightarrow \widehat{\alpha} \notin \mathrm{dom}(, ^{\otimes})$ and $, ^{\otimes} \vdash {<}\Omega{>} :: \Omega$ (DECL-T) $\Rightarrow \alpha \notin \mathrm{dom}(,\ )$ (Lemma 10.7.3) and $\vdash , ^{\otimes}$ valid (Lemma 6.3.5) $\Rightarrow \vdash ,$ valid (induction hypothesis) $\Rightarrow \vdash , , \alpha$ valid (DCL).

Var: Here $,^{\otimes} \vdash \alpha^{\otimes} :: \Omega \Rightarrow \, ,^{\otimes} \vdash \widehat{\alpha}! :: \Omega \Rightarrow \, ,^{\otimes} \vdash \widehat{\alpha} \Rightarrow \, <\Omega>$ or $,^{\otimes} \vdash \widehat{\alpha} \Rightarrow \, <=A::\Omega>$ for some A (C-EXT-O and C-EXT-T) $\Rightarrow \vdash \, ,^{\otimes}$ valid (Lemma 6.3.5) and $\widehat{\alpha} \in \mathrm{dom}(,^{\otimes})$ (Theorem 6.5.6) $\Rightarrow \vdash \, ,$ valid (induction hypothesis) and $\alpha \in \mathrm{dom}(,)$ (Lemma 10.7.3) $\Rightarrow \, , \vdash \alpha$ valid (VAR).

TSum: Here $,^{\otimes} \vdash (\exists\alpha{=}A_1.A_2)^{\otimes} :: \Omega,\ \alpha \notin \mathrm{dom}(,) \Rightarrow \, ,^{\otimes} \vdash \Sigma\widehat{\alpha}{:}<=A_1^{\otimes}::\Omega>.\, A_2^{\otimes} :: \Omega$ and $\widehat{\alpha} \notin \mathrm{dom}(,^{\otimes})$ (Lemma 10.7.3) $\Rightarrow \, ,^{\otimes}, \widehat{\alpha}{:}<=A_1^{\otimes}::\Omega> \vdash A_2^{\otimes} :: \Omega$ (C-DSUM) $\Rightarrow \, ,^{\otimes} \vdash \, <=A_1^{\otimes}::\Omega> :: \Omega$ (Lemma 6.3.5 and DECL-T) and $,^{\otimes}, \widehat{\alpha}{:}<=A_1^{\otimes}::\Omega> \vdash A_2^{\otimes} = [A_1^{\otimes}/\widehat{\alpha}!]A_2^{\otimes} :: \Omega$ (Lemma 10.7.7) $\Rightarrow \, ,^{\otimes} \vdash A_1^{\otimes} :: \Omega$ (C-TRANS) and $,^{\otimes}, \widehat{\alpha}{:}<=A_1^{\otimes}::\Omega> \vdash [A_1^{\otimes}/\widehat{\alpha}!]A_2^{\otimes} :: \Omega$ (Theorem 9.4.3) $\Rightarrow \, ,^{\otimes} \vdash [A_1^{\otimes}/\widehat{\alpha}!]A_2^{\otimes} :: \Omega$ (Theorem 6.5.6, Lemma 10.7.9, and Theorem 9.6.1) $\Rightarrow \, ,^{\otimes} \vdash ([A_1/\alpha]A_2)^{\otimes} :: \Omega$ (Lemma 10.7.3) $\Rightarrow \, , \vdash A_1$ valid and $, \vdash [A_1/\alpha]A_2$ valid (induction hypothesis) $\Rightarrow$ $, \vdash \exists\alpha{=}A_1.A_2$ valid (SUM-T).

□

Because equality in the kernel system between encoded simple types can proceed via transitivity through constructors which have no analog in the simple system (e.g., constructor applications), kernel-system equality derivations cannot be directly converted into the simple system. Instead, the derivations must converted as we go into a form that does not use the transitivity rule; this conversion is accomplished by using the results about how equality acts in a component-wise manner and about how equality relates the intrinsic shapes of encoded types. The size metric is used to ensure that the conversion process terminates.

Lemma [Simple] 10.8.4 (Reduction II: equality)
If $,^{\otimes} \vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega$ *then* $, \vdash A_1 = A_2$.

Proof: Proved by induction on $|A_1|$. By Theorem 9.4.3, Lemma 10.7.4, Lemma 10.3.3, and the definition of intrinsic shapes, we have only the following cases:

Var: Here $,^{\otimes} \vdash \alpha_1^{\otimes} = \alpha_2^{\otimes} :: \Omega \Rightarrow \, ,^{\otimes} \vdash \widehat{\alpha_1}! = \widehat{\alpha_2}! :: \Omega \Rightarrow \, ,^{\otimes} \vdash \widehat{\alpha_1}! :: \Omega$ (Theorem 9.4.3) $\Rightarrow$ $, \vdash \alpha_1$ valid (Lemma 10.8.3).

Hence, for $i \in \{1,2\}$, we have: $,^{\otimes} \vdash \widehat{\alpha_i}! :: \Omega$ (Theorem 9.4.3) $\Rightarrow \widehat{\alpha_i} \in \mathrm{dom}(,^{\otimes})$ (Theorem 6.5.6) $\Rightarrow \alpha_i \in \mathrm{dom}(,)$ (Lemma 10.7.3) $\Rightarrow \widehat{\alpha_i}{:}<\Omega> \in \, ,^{\otimes} \Rightarrow \, ,^{\otimes}(\alpha_i) = \, <\Omega>$ (Lemma 9.4.2) $\Rightarrow \widehat{\alpha_i}! \in C_{\otimes}$ (Lemma 9.5.11) $\Rightarrow \widehat{\alpha_1}! = \widehat{\alpha_2}!$ (Lemma 9.5.12) $\Rightarrow \alpha_1 = \alpha_2 \Rightarrow \, , \vdash \alpha_1 = \alpha_2$ (EQ-VAR).

Neg: $,^{\otimes} \vdash (\neg A_1)^{\otimes} = (\neg A_2)^{\otimes} :: \Omega$
$\Rightarrow \, ,^{\otimes} \vdash \Pi x{:}A_1^{\otimes}.<\Omega> = \Pi x{:}A_2^{\otimes}.<\Omega> :: \Omega,\ x \notin \mathrm{dom}(,^{\otimes})$
$\Rightarrow \, ,^{\otimes} \vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega$ (Lemma 9.4.10) $\Rightarrow \, , \vdash A_1 = A_2$ (induction hypothesis) $\Rightarrow$ $, \vdash \neg A_1 = \neg A_2$ (EQ-NEG).

OSum: $, ^{\otimes} \vdash (\exists\alpha.A_1)^{\otimes} = (\exists\alpha.A_2)^{\otimes} :: \Omega,\ \alpha \notin \mathrm{dom}(,\ )$
$\Rightarrow\ , ^{\otimes} \vdash \Sigma\widehat{\alpha}{:}{<}\Omega{>}.\ A_1^{\otimes} = \Sigma\widehat{\alpha}{:}{<}\Omega{>}.\ A_2^{\otimes} :: \Omega$ and $\widehat{\alpha} \notin \mathrm{dom}(, ^{\otimes})$
(Lemma 10.7.3) $\Rightarrow\ , ^{\otimes}, \widehat{\alpha}{:}{<}\Omega{>} \vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega$ (Lemma 9.4.10)
$\Rightarrow\ , , \alpha \vdash A_1 = A_2$ (induction hypothesis) $\Rightarrow\ ,\ \vdash \exists\alpha.A_1 = \exists\alpha.A_2$ (EQ-SUM-O).

TSum: $, ^{\otimes} \vdash (\exists\alpha{=}A_1.A_2)^{\otimes} = (\exists\alpha{=}A_1'.A_2')^{\otimes} :: \Omega,\ \alpha \notin \mathrm{dom}(,\ )$
$\Rightarrow\ , ^{\otimes} \vdash \Sigma\widehat{\alpha}{:}{<}{=}A_1^{\otimes}{::}\Omega{>}.\ A_2^{\otimes} = \Sigma\widehat{\alpha}{:}{<}{=}A_1'^{\otimes}{::}\Omega{>}.\ A_2'^{\otimes} :: \Omega$ and $\widehat{\alpha} \notin\ , ^{\otimes}$ (Lemma 10.7.3)
$\Rightarrow\ , ^{\otimes} \vdash {<}{=}A_1^{\otimes}{::}\Omega{>} = {<}{=}A_1'^{\otimes}{::}\Omega{>} :: \Omega$ and
$, ^{\otimes}, \widehat{\alpha}{:}{<}{=}A_1^{\otimes}{::}\Omega{>} \vdash A_2^{\otimes} = A_2'^{\otimes} :: \Omega$ (Lemma 9.4.10) $\Rightarrow\ , ^{\otimes} \vdash A_1^{\otimes} = A_1'^{\otimes} :: \Omega$
(Lemma 9.4.10) and $, ^{\otimes}, \widehat{\alpha}{:}{<}{=}A_1^{\otimes}{::}\Omega{>} \vdash [A_1^{\otimes}/\widehat{\alpha}]A_2^{\otimes} = [A_1^{\otimes}/\widehat{\alpha}]A_2'^{\otimes} :: \Omega$
(Theorem 9.4.3, Lemma 10.7.7, E-SYM, and E-TRAN)
$\Rightarrow\ , ^{\otimes} \vdash [A_1^{\otimes}/\widehat{\alpha}]A_2^{\otimes} = [A_1^{\otimes}/\widehat{\alpha}]A_2'^{\otimes} :: \Omega$ (Theorem 9.4.3, Theorem 6.5.6,
Lemma 10.7.9, and Theorem 9.6.1)
$\Rightarrow\ , ^{\otimes} \vdash ([A_1/\alpha]A_2)^{\otimes} = ([A_1/\alpha]A_2')^{\otimes} :: \Omega$ (Lemma 10.7.3) $\Rightarrow\ ,\ \vdash A_1 = A_1'$ and
$,\ \vdash [A_1/\alpha]A_2 = [A_1/\alpha]A_2'$ (induction hypothesis)
$\Rightarrow\ ,\ \vdash \exists\alpha{=}A_1.A_2 = \exists\alpha{=}A_1'.A_2'$ (EQ-SUM-T).

□

Some useful properties about the simple equality and subtyping relations can be transfered from the kernel system using the results so far:

Theorem [Simple] 10.8.5 (Equality properties)

1. *If* $,\ \vdash A$ *valid then* $,\ \vdash A = A$.
2. *If* $,\ \vdash A_1 = A_2$ *then* $,\ \vdash A_2 = A_1$.
3. *If* $,\ \vdash A_1 = A_2$ *and* $,\ \vdash A_2 = A_3$ *then* $,\ \vdash A_1 = A_3$.

Proof: Each part is proved the same way: first convert to the kernel system using Lemma 10.8.1, second apply the relevant kernel rule (E-REFL, E-SYM, and E-TRAN respectively), and third, transfer the result back to the simple system using Lemma 10.8.3 or Lemma 10.8.4. □

Corollary [Simple] 10.8.6 (Reflexivity) *If* $,\ \vdash A$ *valid then* $,\ \vdash A \leq A$.

Proof: Follows immediately from Theorem 10.8.5 and Lemma 10.6.12. □

Theorem [Simple] 10.8.7 (Structural property II)

1. *If* $, \vdash A_1 = A_2$ *then* $, \vdash A_1$ *valid and* $, \vdash A_2$ *valid.*
2. *If* $, \vdash A_1 \leq A_2$ *then* $, \vdash A_1$ *valid and* $, \vdash A_2$ *valid.*

Proof: Proved by first converting to the kernel system using Lemma 10.8.1, second applying either Theorem 9.4.3 or Lemma 10.2.2, and third, transferring the result back to the simple system using Lemma 10.8.3. □

A similar problem to that of the equality judgment case arises with the subtyping judgment case. Unfortunately, because of the SU-NEG and SU-FOR rules, the simple-type size metric cannot be used to ensure termination here, forcing a different proof strategy to be used. My solution to this problem involves taking advantage of the fact that kernel-system subtyping is defined (see Section 10.2) in terms of a series of equality and one-step subtyping steps joined by transitivity.

I have already shown that kernel-system equality steps can be converted to simple subtyping steps (Lemmas 10.8.4 and 10.6.12). I shall show next that one-step subtyping can also be converted to simple subtyping. Because the one-step subtyping relation lacks any transitivity rule, the proof of this fact is fairly straightforward. By showing that simple subtyping is transitive, I shall then have that any kernel-system subtyping judgment derivation can be converted to a simple subtyping judgment derivation by converting its steps one at a time to simple subtyping and then combining the resulting steps via transitivity.

Lemma [Simple] 10.8.8 (Reduction II: one-step subtyping)
If $,^{\otimes} \vdash A_1^{\otimes} \lhd A_2^{\otimes}$ *then* $, \vdash A_1 \leq A_2$.

Proof: Proved by induction on the number of applications of the one-step subtyping rules needed to derive $,^{\otimes} \vdash A_1^{\otimes} \lhd A_2^{\otimes}$. Interesting cases:

O-REFL: Here $,^{\otimes} \vdash A^{\otimes} \lhd A^{\otimes}$ derived via O-REFL from $,^{\otimes} \vdash A^{\otimes} :: \Omega \Rightarrow , \vdash A$ valid (Lemma 10.8.3) $\Rightarrow , \vdash A \leq A$ (Corollary 10.8.6)

O-DFUN: Here $,^{\otimes} \vdash (\neg A_1)^{\otimes} \lhd (\neg A_2)^{\otimes}$ derived via O-DFUN from $,^{\otimes} \vdash A_2^{\otimes} \lhd A_1^{\otimes} \Rightarrow , \vdash A_2 \leq A_1$ (induction hypothesis) $\Rightarrow , \vdash \neg A_1 \leq \neg A_2$ (SU-NEG).

O-DSUM I: Here $,^{\otimes} \vdash (\exists\alpha.A_1)^{\otimes} \lhd (\exists\alpha.A_2)^{\otimes}$, $\alpha \notin \text{dom}(,)$, derived via O-DSUM from $,^{\otimes} \vdash <\Omega> \lhd <\Omega>$, $,^{\otimes}, \hat{\alpha}{:}<\Omega> \vdash A_1^{\otimes} \lhd A_2^{\otimes}$, and $,^{\otimes}, \hat{\alpha}{:}<\Omega> \vdash A_2^{\otimes} :: \Omega$ $\Rightarrow , , \alpha \vdash A_1 \leq A_2$ (induction hypothesis) $\Rightarrow , \vdash \exists\alpha.A_1 \leq \exists\alpha.A_2$ (SU-SUM-O).

O-DSUM II: Here $,^{\otimes} \vdash (\exists\alpha{=}A_1'.A_1)^{\otimes} \lhd (\exists\alpha.A_2)^{\otimes}$, $\alpha \notin \mathrm{dom}(,)$, derived via O-DSUM from $,^{\otimes} \vdash \,<\!=\!A_1'^{\otimes}{::}\Omega\!> \lhd \,<\!\Omega\!>$, $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_1'^{\otimes}{::}\Omega\!> \vdash A_1^{\otimes} \lhd A_2^{\otimes}$, and $,^{\otimes}, \hat{\alpha}{:}<\!\Omega\!> \vdash A_2^{\otimes} :: \Omega \Rightarrow\ , , \alpha \vdash A_2$ valid (Lemma 10.8.3) and $,^{\otimes} \vdash A_1'^{\otimes} :: \Omega$ (O-FORGET) and $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_1'^{\otimes}{::}\Omega\!> \vdash [A_1'^{\otimes}/\hat{\alpha}!]A_1^{\otimes} \lhd [A_1'^{\otimes}/\hat{\alpha}!]A_2^{\otimes}$ (Lemma 10.7.8) $\Rightarrow\ , \vdash A_1'$ valid (Lemma 10.8.3) and $,^{\otimes} \vdash ([A_1'/\alpha]A_1)^{\otimes} \lhd ([A_1'/\alpha]A_2)^{\otimes}$ (Lemma 10.1.4 and Lemma 10.7.3).

Since by Lemma 10.7.8 and by inspection of the proof of Lemma 10.1.4, $,^{\otimes} \vdash ([A_1'/\alpha]A_1)^{\otimes} \lhd ([A_1'/\alpha]A_2)^{\otimes}$ uses the same number of applications of the one-step subtyping rules as does $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_1'^{\otimes}{::}\Omega\!> \vdash A_1^{\otimes} \lhd A_2^{\otimes}$, a sub-derivation of the original derivation, we can apply the induction hypothesis to get $, \vdash [A_1'/\alpha]A_1 \leq [A_1'/\alpha]A_2 \Rightarrow\ , \vdash \exists\alpha{=}A_1'.A_1 \leq \exists\alpha.A_2$ (SU-FOR).

O-DSUM III: Here $,^{\otimes} \vdash (\exists\alpha{=}A_1'.A_1)^{\otimes} \lhd (\exists\alpha{=}A_2'.A_2)^{\otimes}$, $\alpha \notin \mathrm{dom}(,)$, derived via O-DSUM from $,^{\otimes} \vdash \,<\!=\!A_1'^{\otimes}{::}\Omega\!> \lhd \,<\!=\!A_2'^{\otimes}{::}\Omega\!>$, $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_1'^{\otimes}{::}\Omega\!> \vdash A_1^{\otimes} \lhd A_2^{\otimes}$, and $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_2'^{\otimes}{::}\Omega\!> \vdash A_2^{\otimes} :: \Omega \Rightarrow A_1'^{\otimes} = A_2'^{\otimes}$ and $,^{\otimes} \vdash \,<\!=\!A_1'^{\otimes}{::}\Omega\!> :: \Omega$ (O-REFL) and $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_1'^{\otimes}{::}\Omega\!> \vdash [A_1'^{\otimes}/\hat{\alpha}!]A_1^{\otimes} \lhd [A_1'^{\otimes}/\hat{\alpha}!]A_2^{\otimes}$ (Lemma 10.7.8) $\Rightarrow\ ,^{\otimes} \vdash A_1'^{\otimes} :: \Omega$ (C-TRANS) and $,^{\otimes} \vdash ([A_1'/\alpha]A_1)^{\otimes} \lhd ([A_1'/\alpha]A_2)^{\otimes}$ (Lemma 10.1.4 and Lemma 10.7.3) $\Rightarrow\ ,^{\otimes} \vdash A_1'^{\otimes} = A_2'^{\otimes} :: \Omega$ (E-REFL) $\Rightarrow\ , \vdash A_1' = A_2'$ (Lemma 10.8.4).

Since by Lemma 10.7.8 and by inspection of the proof of Lemma 10.1.4, $,^{\otimes} \vdash ([A_1'/\alpha]A_1)^{\otimes} \lhd ([A_1'/\alpha]A_2)^{\otimes}$ uses the same number of applications of the one-step subtyping rules as does $,^{\otimes}, \hat{\alpha}{:}<\!=\!A_1'^{\otimes}{::}\Omega\!> \vdash A_1^{\otimes} \lhd A_2^{\otimes}$, a sub-derivation of the original derivation, we can apply the induction hypothesis to get $, \vdash [A_1'/\alpha]A_1 \leq [A_1'/\alpha]A_2 \Rightarrow\ , \vdash \exists\alpha{=}A_1'.A_1 \leq \exists\alpha{=}A_2'.A_2$ (SU-SUM-T).

□

In order to prove transitivity for simple subtyping, we shall first need to prove a lemma that if $, \vdash A = A'$ and $, \vdash [A/\alpha]A_1 \leq [A/\alpha]A_2$ by an extended derivation, then $, \vdash [A'/\alpha]A_1 \leq [A'/\alpha]A_2$ by an extended derivation of equal size. This result is needed to handle the case where we have that $, \vdash \exists\alpha{=}A_1.A_1' \leq \exists\alpha{=}A_2.A_2'$ and $, \vdash \exists\alpha{=}A_2.A_2' \leq \exists\alpha{=}A_3.A_3'$, both by extended derivations via the SU-SUM-T rule. By SU-SUM-T, we also have that $, \vdash A_1 = A_2$, $, \vdash [A_1/\alpha]A_1' \leq [A_1/\alpha]A_2'$ and $, \vdash [A_2/\alpha]A_2' \leq [A_2/\alpha]A_3'$. We need to show that $, \vdash [A_1/\alpha]A_1' \leq [A_1/\alpha]A_3'$ so we can apply the SU-SUM-T rule to get that $, \vdash \exists\alpha{=}A_1.A_1' \leq \exists\alpha{=}A_3.A_3'$.

The lemma will allow us to deduce that $, \vdash [A_1/\alpha]A_2' \leq [A_1/\alpha]A_3'$. The needed result would then follow from transitivity if available. In order to use transitivity (the induction hypothesis) here, though, we need to know that we are dealing with a smaller problem. We can arrange this by inducting on the size of the extended derivations involved and taking advantage of the fact that extended derivation size is unchanged by this lemma.

Lemma [Simple] 10.8.9 *If* $, \vdash [A_1/\alpha]A$ *valid and* $, \vdash A_1 = A_2$ *then* $, \vdash [A_1/\alpha]A = [A_2/\alpha]A$.

Proof: Proved by structural induction on A. □

Lemma [Simple] 10.8.10 *If* $, \vdash A = A'$ *and* $, \vdash [A/\alpha]A_1 = [A/\alpha]A_2$ *then* $, \vdash [A'/\alpha]A_1 = [A'/\alpha]A_2$.

Proof: By Theorem 10.8.7, $, \vdash [A/\alpha]A_1$ valid and $, \vdash [A/\alpha]A_2$ valid $\Rightarrow , \vdash [A/\alpha]A_1 = [A'/\alpha]A_1$ and $, \vdash [A/\alpha]A_2 = [A'/\alpha]A_2$ (Lemma 10.8.9) $\Rightarrow , \vdash [A'/\alpha]A_1 = [A/\alpha]A_1$ (Theorem 10.8.5) $\Rightarrow , \vdash [A'/\alpha]A_1 = [A'/\alpha]A_2$ (Theorem 10.8.5).
□

Lemma [Simple] 10.8.11 *Suppose* $, \vdash A_1 \leq A_2$ *by an extended derivation of size* s. *Then:*

1. *If* $, \vdash A_0 = A_1$ *then* $, \vdash A_0 \leq A_2$ *by an extended derivation of size* s.
2. *If* $, \vdash A_2 = A_3$ *then* $, \vdash A_1 \leq A_3$ *by an extended derivation of size* s.
3. *If* $A_1 = [A/\alpha]A'_1$, $A_2 = [A/\alpha]A'_2$, *and* $, \vdash A = A'$ *then* $, \vdash [A'/\alpha]A'_1 \leq [A'/\alpha]A'_2$ *by an extended derivation of size* s.

Proof: Proved by simultaneous induction on s, with the proviso that part three may also call parts one and two with a derivation of the same size. Interesting cases:

SU-NEG 1: Here $, \vdash \neg A_0 = \neg A_1$ and $, \vdash \neg A_1 \leq \neg A_2$ via an extended derivation starting with rule SU-NEG from $, \vdash A_2 \leq A_1$ (with extended size $s \Leftrightarrow 1$) $\Rightarrow , \vdash A_0 = A_1$ (EQ-NEG) $\Rightarrow , \vdash A_1 = A_0$ (Theorem 10.8.5) $\Rightarrow , \vdash A_2 \leq A_0$ (with extended size $s \Leftrightarrow 1$) (induction via part two) $\Rightarrow , \vdash \neg A_0 \leq \neg A_2$ (with extended size s) (SU-NEG).

SU-EQ 1: Here $, \vdash A_0 = A_1$ and $, \vdash A_1 \leq A_2$ via an extended derivation of size 1 starting with rule SU-EQ from $, \vdash A_1 = A_2 \Rightarrow , \vdash A_0 = A_2$ (Theorem 10.8.5) $\Rightarrow , \vdash A_0 \leq A_2$ (with extended size 1) (SU-EQ).

SU-SUM-T 1: Here $, \vdash \exists\alpha{=}A_0.A'_0 = \exists\alpha{=}A_1.A'_1$ and $, \vdash \exists\alpha{=}A_1.A'_1 \leq \exists\alpha{=}A_2.A'_2$ via an extended derivation starting with rule SU-SUM-T from $, \vdash A_1 = A_2$ and $, \vdash [A_1/\alpha]A'_1 \leq [A_1/\alpha]A'_2$ (with extended size $s \Leftrightarrow 1$) $\Rightarrow , \vdash A_0 = A_1$ and $, \vdash [A_0/\alpha]A'_0 = [A_0/\alpha]A'_1$ (EQ-SUM-T) $\Rightarrow , \vdash A_0 = A_2$ (Theorem 10.8.5) and $, \vdash [A_0/\alpha]A'_1 \leq [A_0/\alpha]A'_2$ (with extended size $s \Leftrightarrow 1$) (induction via part three) $\Rightarrow$ $, \vdash [A_0/\alpha]A'_0 \leq [A_0/\alpha]A'_2$ (with extended size $s \Leftrightarrow 1$) (induction via part one and Theorem 10.8.5) $\Rightarrow , \vdash \exists\alpha{=}A_0.A'_0 \leq \exists\alpha{=}A_2.A'_2$ (with extended size s) (SU-SUM-T).

SU-SUM-T 2: Here $, \vdash \exists\alpha{=}A_2.A'_2 = \exists\alpha{=}A_3.A'_3$ and $, \vdash \exists\alpha{=}A_1.A'_1 \leq \exists\alpha{=}A_2.A'_2$ via an extended derivation starting with rule SU-SUM-T from $, \vdash A_1 = A_2$ and $, \vdash [A_1/\alpha]A'_1 \leq [A_1/\alpha]A'_2$ (with extended size $s{\Leftrightarrow}1$) $\Rightarrow\ , \vdash A_2 = A_3$ and $, \vdash [A_2/\alpha]A'_2 = [A_2/\alpha]A'_3$ (EQ-SUM-T) $, \vdash A_1 = A_3$ (Theorem 10.8.5) and $, \vdash [A_1/\alpha]A'_2 = [A_1/\alpha]A'_3$ (Lemma 10.8.10) $\Rightarrow\ , \vdash [A_1/\alpha]A'_1 \leq [A_1/\alpha]A'_3$ (with extended size $s{\Leftrightarrow}1$) (induction via part two) $\Rightarrow\ , \vdash \exists\alpha{=}A_1.A'_1 \leq \exists\alpha{=}A_3.A'_3$ (with extended size s) (SU-SUM-T).

SU-FOR 2: Here $, \vdash \exists\alpha.A'_2 = \exists\alpha.A'_3$ and $, \vdash \exists\alpha{=}A_1.A'_1 \leq \exists\alpha.A'_2$ via an extended derivation starting with rule SU-FOR from $, \vdash A_1$ valid, $, ,\alpha \vdash A'_2$ valid, and $, \vdash [A_1/\alpha]A'_1 \leq [A_1/\alpha]A'_2$ (with extended size $s{\Leftrightarrow}1$) $\Rightarrow\ , ,\alpha \vdash A'_2 = A'_3$ (EQ-SUM-O) $\Rightarrow\ , ,\alpha \vdash A'_3$ valid (Theorem 10.8.7) and $, \vdash [A_1/\alpha]A'_2 = [A_1/\alpha]A'_3$ (Corollary 10.6.16) $\Rightarrow\ , \vdash [A_1/\alpha]A'_1 \leq [A_1/\alpha]A'_3$ (with extended size $s{\Leftrightarrow}1$) (induction via part two) $\Rightarrow\ , \vdash \exists\alpha{=}A_1.A'_1 \leq \exists\alpha.A'_3$ (with extended size s) (SU-FOR).

PART 3: Here $, \vdash A = A'$ and $, \vdash [A/\alpha]A'_1 \leq [A/\alpha]A'_2$ by an extended derivation of size $s \Rightarrow\ , \vdash [A/\alpha]A'_1$ valid and $, \vdash [A/\alpha]A'_2$ valid (Theorem 10.8.7) $\Rightarrow\ , \vdash [A/\alpha]A'_1 = [A'/\alpha]A'_1$ and $, \vdash [A/\alpha]A'_2 = [A'/\alpha]A'_2$ (Lemma 10.8.9) $\Rightarrow\ , \vdash [A'/\alpha]A'_1 = [A/\alpha]A'_1$ (Theorem 10.8.5) $\Rightarrow\ , \vdash [A'/\alpha]A'_1 \leq [A/\alpha]A'_2$ (with extended size s) (induction via part one) $\Rightarrow\ , \vdash [A'/\alpha]A'_1 \leq [A'/\alpha]A'_2$ (with extended size s) (induction via part two).

□

Theorem [Simple] 10.8.12 (Transitivity)
If $, \vdash A_1 \leq A_2$ *by an extended derivation and* $, \vdash A_2 \leq A_3$ *by an extended derivation then* $, \vdash A_1 \leq A_3$.

Proof: Proved by induction on the sum of the sizes of the two extended derivations. Interesting cases:

NEG: Here $, \vdash \neg A_1 \leq \neg A_2$ by an extended derivation via SU-NEG from $, \vdash A_2 \leq A_1$ and $, \vdash \neg A_2 \leq \neg A_3$ by an extended derivation via SU-NEG from $, \vdash A_3 \leq A_2 \Rightarrow\ , \vdash A_3 \leq A_1$ (induction — the sum of sizes is unchanged by swapping the derivations) $\Rightarrow\ , \vdash \neg A_1 \leq \neg A_3$ (SU-NEG).

EQ L: Here $, \vdash A_1 \leq A_2$ by an extended derivation via SU-EQ from $, \vdash A_1 = A_2$ and $, \vdash A_2 \leq A_3$ by an extended derivation $\Rightarrow\ , \vdash A_1 \leq A_3$ (Lemma 10.8.11).

TSUM: Here $, \vdash \exists\alpha{=}A_1.A'_1 \leq \exists\alpha{=}A_2.A'_2$ by an extended derivation via SU-SUM-T from $, \vdash A_1 = A_2$ and $, \vdash [A_1/\alpha]A'_1 \leq [A_1/\alpha]A'_2$ and $, \vdash \exists\alpha{=}A_2.A'_2 \leq \exists\alpha{=}A_3.A'_3$ by an extended derivation via SU-SUM-T from

$, \vdash A_2 = A_3$ and $, \vdash [A_2/\alpha]A_2' \leq [A_2/\alpha]A_3'$ (with extended size s) $\Rightarrow\ , \vdash A_1 = A_3$ (Theorem 10.8.5) and $, \vdash [A_1/\alpha]A_2' \leq [A_1/\alpha]A_3'$ (with extended size s) (Lemma 10.8.11) $\Rightarrow\ , \vdash [A_1/\alpha]A_1' \leq [A_1/\alpha]A_3'$ (induction)
$\Rightarrow\ , \vdash \exists\alpha{=}A_1.A_1' \leq \exists\alpha{=}A_3.A_3'$ (SU-SUM-T).

FOR L: Here $, \vdash \exists\alpha{=}A_1.A_1' \leq \exists\alpha.A_2'$ by an extended derivation via SU-FOR from $, \vdash A_1$ valid, $, , \alpha \vdash A_2'$ valid, and $, \vdash [A_1/\alpha]A_1' \leq [A_1/\alpha]A_2'$ and $, \vdash \exists\alpha.A_2' \leq \exists\alpha.A_3'$ by an extended derivation via SU-SUM-O from $, , \alpha \vdash A_2' \leq A_3'$ (with extended size s) $\Rightarrow\ , , \alpha \vdash A_3'$ valid (Theorem 10.8.7) and $, \vdash [A_1/\alpha]A_2' \leq [A_1/\alpha]A_3'$ (with extended size s) (Corollary 10.6.16)
$\Rightarrow\ , \vdash [A_1/\alpha]A_1' \leq [A_1/\alpha]A_3'$ (induction) $\Rightarrow\ , \vdash \exists\alpha{=}A_1.A_1' \leq \exists\alpha.A_3'$ (SU-FOR).

□

Lemma [Simple] 10.8.13 (Reduction II: subtyping)
If $,^{\otimes} \vdash A_1^{\otimes} \leq A_2^{\otimes}$ *then* $, \vdash A_1 \leq A_2$.

Proof: Proved by structural induction on the derivation. The S-EQ case is handled by Lemma 10.8.4 and Lemma 10.6.12, the S-ONE case is handled by Lemma 10.8.8, and the S-TRAN case is handled by Theorem 10.8.12. □

Thus, I have shown that any simple judgment may be decided by first encoding it as a kernel-system judgment and then deciding the encoded judgment:

Theorem [Simple] 10.8.14 (Reduction III)

1. $\vdash ,$ *valid iff* $\vdash ,^{\otimes}$ *valid.*
2. $, \vdash A$ *valid iff* $,^{\otimes} \vdash A^{\otimes} :: \Omega$.
3. $, \vdash A_1 = A_2$ *iff* $,^{\otimes} \vdash A_1^{\otimes} = A_2^{\otimes} :: \Omega$.
4. $, \vdash A_1 \leq A_2$ *iff* $,^{\otimes} \vdash A_1^{\otimes} \leq A_2^{\otimes}$.

Proof: Follows immediately from Lemmas 10.8.1, 10.8.3, 10.8.4, and 10.8.13. □

As consequences of this, the simple validity and equality judgments are decidable and the simple subtyping problem reduces to the kernel-system subtyping problem. This last fact means that if the simple subtyping problem is undecidable, then the kernel-system subtyping problem must be undecidable as well.

Corollary [Simple] 10.8.15 (Problem reduction)

1. *The $\vdash ,$ valid judgment is decidable.*
2. *The $, \vdash A$ valid judgment is decidable.*
3. *The $, \vdash A_1 = A_2$ judgment is decidable.*
4. *If the $, \vdash A_1 \leq A_2$ judgment is undecidable, then the kernel system subtyping problem is undecidable.*

Proof: Follows from Theorem 10.8.14, Theorem 9.4.11, and the fact that the encoding can be done by an algorithm. □

10.9 Undecidability

In this section I prove that the simple subtyping problem, and hence the kernel-system subtyping problem, is undecidable. All judgments in this section are from the simple system. First, it is worth noting that in the absence of the SU-FOR rule, simple subtyping is decidable:

Lemma [Simple] 10.9.1

1. *If $|A_1| = |A_2|$ then $|[A_1/\alpha]A| = |[A_2/\alpha]A|$.*
2. *If $, \vdash A_1 = A_2$ then $|A_1| = |A_2|$.*

Proof: Proved sequentially by induction on A and the derivation of $, \vdash A_1 = A_2$ respectively. □

Theorem [Simple] 10.9.2 (Decidability)
If the SU-FOR rule is removed, then simple subtyping is decidable.

Proof: Inspection of the other SU rules shows that each of them strictly decreases the following non-negative measure, so the simple syntax-directed procedure always terminates in this case: $|, \vdash A_1 \leq A_2| = |A_1| + |A_2|$. (Lemma 10.9.1 is needed to handle the SU-SUM-T case; the decidability of the other judgments' procedures is handled by

Corollary 10.8.15.) $\square$

Note that use of the SU-FOR rule does not decrease this measure and in fact can increase it because the type on the right side can grow without limit in the recursive call. This fact can be used to construct examples that cause the simple syntax-directed procedure for checking simple subtyping to loop. For example, consider the following definitions:

$$\begin{array}{rcl} P(A) & = & \exists\alpha{=}A.\neg A \quad (\alpha \notin \mathrm{FV}(A)) \\ G_\alpha(A) & = & \exists\alpha.\neg A \end{array}$$

An example which causes cyclic behavior under the empty assignment is then as follows:

$$\begin{array}{rlcl} & P(G_\alpha(P(\alpha))) & \leq & G_\alpha(P(\alpha)) \\ = & \exists\alpha'{=}G_\alpha(P(\alpha)).\neg G_\alpha(P(\alpha)) & \leq & \exists\alpha'.\neg P(\alpha') \\ \Rightarrow & [G_\alpha(P(\alpha))/\alpha'](\neg G_\alpha(P(\alpha))) & \leq & [G_\alpha(P(\alpha))/\alpha'](\neg P(\alpha')) \\ = & \neg G_\alpha(P(\alpha)) & \leq & \neg P(G_\alpha(P(\alpha))) \\ \Rightarrow & P(G_\alpha(P(\alpha))) & \leq & G_\alpha(P(\alpha)) \\ & & \vdots & \end{array}$$

Pierce's proof of the undecidability of $F_\leq$ subtyping has two steps. First, he introduces a new kind of machine called a *row machine* and shows that the halting problem for these machines is undecidable by reducing the two-counter Turing machine halting problem, a known undecidable problem, to it. Second, he shows that $F_\leq$ subtyping can be used to decide rowing machine halting problems.

Row machines come in a variety of positive widths; the syntax for n-width rowing machines is as follows:

Definition [Rowing] 10.9.3 (Syntax of n-width row machines)

$$\begin{array}{lll} \textit{Rows} & \rho & ::= \;\; \alpha \mid \lambda\alpha_1,\ldots,\alpha_n.\, m \mid \textit{HALT} \\ \textit{Machines} & m & ::= \;\; <\rho_1,\ldots,\rho_n> \end{array}$$

Here, the metavariable α ranges over *row* variables. Equivalence of rows and row machines is defined as usual (α-conversion). The definition of free row variables (FV(⇔)) and row substitution ($[\rho/\alpha]$⇔) on rows and row machines are the obvious ones:

Definition [Rowing] 10.9.4 (Free row variables)

$$\begin{array}{rcl} FV(\alpha) & = & \{\alpha\} \\ FV(\lambda\alpha_1,\ldots,\alpha_n.\, m) & = & FV(m) \Leftrightarrow \{\alpha_1,\ldots,\alpha_n\} \\ FV(\textit{HALT}) & = & \emptyset \\ & & \\ FV(<\rho_1,\ldots,\rho_n>) & = & FV(\rho_1) \cup \cdots \cup FV(\rho_n) \end{array}$$

Definition [Rowing] 10.9.5 (Row substitution)

$$\begin{array}{rcll}
[\rho/\alpha]\alpha & = & \rho & \\
[\rho/\alpha]\alpha' & = & \alpha' & (\alpha \neq \alpha') \\
[\rho/\alpha]\lambda\alpha'_1,\ldots,\alpha'_n.\, m & = & \lambda\alpha'_1,\ldots,\alpha'_n.\, [\rho/\alpha]m & (\forall i.\ \alpha'_i \neq \alpha, \alpha'_i \notin FV(\rho)) \\
[\rho/\alpha]HALT & = & HALT & \\
 & & & \\
[\rho/\alpha]{<}\rho_1,\ldots,\rho_n{>} & = & {<}[\rho/\alpha]\rho_1,\ldots,[\rho/\alpha]\rho_n{>} &
\end{array}$$

Row machines of width n are elaborated using the following rewriting relation:

Definition [Rowing] 10.9.6

$$\frac{\rho_1 = \lambda\alpha_1,\ldots,\alpha_n.\, m' \qquad m'' = [\rho_n/\alpha_n]\ldots[\rho_1/\alpha_1]m'}{{<}\rho_1,\ldots,\rho_n{>} \rightarrow m''} \qquad \text{(ROW)}$$

Only closed row machines (FV(m) = $\emptyset$) are elaborated; the rewriting relation preserves the closed property:

Lemma [Rowing] 10.9.7 *If $FV(m) = \emptyset$ and $m \rightarrow m'$ then $FV(m') = \emptyset$.*

Proof: First show by induction on m that if FV(ρ) = $\emptyset$ then FV($[\rho/\alpha]m$) $\subseteq$ (FV(m) $\Leftrightarrow$ $\{\alpha\}$). The result then follows easily from the definitions of row machine rewriting, free row variables, and row substitution. □

The basic idea of row machine elaboration is that a row machine of width n contains n registers, each of which may contain a row. The row machine ${<}\rho_1,\ldots,\rho_n{>}$ represents a machine who's first register holds ρ_1, who's second register holds ρ_2, and so on. The first register of a row machine holds its *program counter* (PC). To move to the next state, the PC is used as a template to construct the new contents of each of the registers from the current contents of all of the registers (including the PC). A row machine halts when its PC becomes HALT:

Definition [Rowing] 10.9.8 *The closed n-width row machine m is said to halt iff $\exists\rho_2,\ldots,\rho_n.\ m \rightarrow^* {<}HALT,\rho_2,\ldots,\rho_n{>}$.*

A closed row machine that never halts will run forever:

Lemma [Rowing] 10.9.9
If $FV(m) = \emptyset$ and $\forall\rho_2,\ldots,\rho_n.\ m \neq {<}HALT,\rho_2,\ldots,\rho_n{>}$ then $\exists m'.\ m \rightarrow m'$.

Proof: Let $m = <\rho_1, \ldots, \rho_n> \Rightarrow \mathrm{FV}(\rho_1) = \emptyset$ (definition of free row variables) and $\rho_1 \neq \mathrm{HALT}$ (given) $\Rightarrow \rho_1 = \lambda\alpha_1, \ldots, \alpha_n.\, m''$ for some $\alpha_1, \ldots, \alpha_n$, and $m'' \Rightarrow \exists m'.\, m \to m'$ (ROW). □

More discussion of row machines, including examples, can be found in Section 6.5 of Pierce's thesis [51]. The elaboration function given there differs slightly from the one here because it uses simultaneous substitution where I have used iterated substitution; the definitions can easily be seen to coincide on closed row machines however. In Section 6.8 of his thesis, Pierce proves the following theorem:

Theorem [Rowing] 10.9.10 (Undecidability)
The halting problem for closed row machines is undecidable. (The width may vary from problem instance to problem instance.)

By making some slight modifications, Pierce's encoding of row machines into $F_\leq$ subtyping can be turned into an encoding of row machines into simple subtyping. The following encoding (F(⇔)) of width n rows and row machines into simple types is designed so that a closed n-width row machine m halts iff the simple judgment $\bullet \vdash \mathrm{F}(m) \leq \sigma$ holds (σ is defined below):

Definition [Rowing] 10.9.11 (Encoding row machines of width n)

$$\begin{array}{lcl}
\sigma & = & \exists\beta.\exists\alpha'_1.\cdots\exists\alpha'_n.\neg\exists\alpha''_1{=}\alpha'_1.\cdots\exists\alpha''_n{=}\alpha'_n.\neg\beta \\
\\
F(\alpha) & = & \alpha \\
F(\lambda\alpha_1,\ldots,\alpha_n.\, m) & = & \exists\alpha_1.\cdots\exists\alpha_n.\neg F(m) \\
F(HALT) & = & \exists\alpha_1.\cdots\exists\alpha_n.\neg\sigma \\
\\
F(<\rho_1,\ldots,\rho_n>) & = & \exists\beta{=}\sigma.\exists\alpha'_1{=}F(\rho_1).\cdots\exists\alpha'_n{=}F(\rho_n).\neg\alpha'_1, \\
 & & (\{\beta,\alpha'_1,\ldots,\alpha'_n\} \cap FV(<\rho_1,\ldots,\rho_n>) = \emptyset)
\end{array}$$

Where $|\{\beta, \alpha'_1, \ldots, \alpha'_n, \alpha''_1, \ldots, \alpha''_n\}| = 2n+1$.

The key differences from Pierce's encoding (see Section 6.6 of his thesis) are as follows:

- Use of $, \vdash \exists\alpha{=}A.A_1 \leq \exists\alpha.A_2$ instead of $\forall\alpha.A_2 \leq \forall\alpha{\leq}A.A_1$.
- Use of reflexivity to halt computation instead of the FTOP rule. (Compare the two definitions of F(HALT))

The proof that the encoding is correct is straightforward:

Lemma [Rowing] 10.9.12 (Encoding properties)

1. $FV(\sigma) = \emptyset$
2. $FV(F(\rho)) = FV(\rho)$ *and* $FV(F(m)) = FV(m)$
3. $F([\rho'/\alpha]\rho) = [F(\rho')/\alpha]F(\rho)$ *and* $F([\rho'/\alpha]m) = [F(\rho')/\alpha]F(m)$

Proof: The first part is proved by inspecting the definition of σ. The remaining two parts are proved by simultaneous induction on ρ and m. □

Lemma [Mixed] 10.9.13 (Encoding validity)

1. $\bullet \vdash \sigma$ *valid*
2. *If* $FV(\rho) \subseteq dom(\,,\,)$ *and* $\vdash ,\;$ *valid then* $,\; \vdash F(\rho)$ *valid.*
3. *If* $FV(m) \subseteq dom(\,,\,)$ *and* $\vdash ,\;$ *valid then* $,\; \vdash F(m)$ *valid.*

(Here , is a simple assignment; all other variables are from the row system.)

Proof: The first part is proved by using VAR, NEG, SUM-O, and SUM-T. The remaining parts are proved by simultaneous structural induction on ρ and m, using Lemmas 10.9.12 and 10.6.11 as needed. □

Lemma [Rowing] 10.9.14 *If* $FV(<HALT, \rho_2, \ldots, \rho_n>) = \emptyset$ *then* $\bullet \vdash F(<HALT, \rho_2, \ldots, \rho_n>) \leq \sigma$.

Proof: By Corollary 10.8.6, NEG, and Lemma 10.9.13, $\bullet \vdash \neg\sigma \leq \neg\sigma$
$\Rightarrow \bullet \vdash \exists\alpha''_1{=}\mathrm{F(HALT)}.\exists\alpha''_2{=}\mathrm{F}(\rho_2).\cdots\exists\alpha''_n{=}\mathrm{F}(\rho_2).\neg\sigma \leq \exists\alpha_1.\cdots\exists\alpha_n.\neg\sigma$
(Lemma 10.9.12, 10.9.13, SUM-O, SU-FOR, and the given)
$\Rightarrow \bullet \vdash \exists\alpha''_1{=}\mathrm{F(HALT)}.\exists\alpha''_2{=}\mathrm{F}(\rho_2).\cdots\exists\alpha''_n{=}\mathrm{F}(\rho_2).\neg\sigma \leq \mathrm{F(HALT)}$
$\Rightarrow \bullet \vdash \neg\mathrm{F(HALT)} \leq \neg\exists\alpha''_1{=}\mathrm{F(HALT)}.\exists\alpha''_2{=}\mathrm{F}(\rho_2).\cdots\exists\alpha''_n{=}\mathrm{F}(\rho_2).\neg\sigma$ (SU-NEG)
$\Rightarrow \bullet \vdash \exists\beta{=}\sigma.\exists\alpha'_1{=}\mathrm{F(HALT)}.\exists\alpha'_2{=}\mathrm{F}(\rho_2).\cdots\exists\alpha'_n{=}\mathrm{F}(\rho_n).\neg\alpha'_1 \leq$
$\exists\beta.\exists\alpha'_1.\cdots\exists\alpha'_n.\neg\exists\alpha''_1{=}\alpha'_1.\cdots\exists\alpha''_n{=}\alpha'_n.\neg\beta$ where $|\{\beta, \alpha'_1, \ldots, \alpha'_n, \alpha''_1, \ldots, \alpha''_n\}| = 2n+1$
(Lemma 10.9.12, 10.9.13, SUM-O, SU-FOR, and the given)
$\Rightarrow \bullet \vdash \mathrm{F}(<\mathrm{HALT}, \rho_2, \ldots, \rho_n>) \leq \sigma$. □

Lemma [Rowing] 10.9.15
Suppose $m = <\rho_1, \ldots, \rho_n>$, $FV(m) = \emptyset$, and $\rho_1 \neq HALT$. Then:

1. *If $m \rightarrow m'$ and $\bullet \vdash F(m') \leq \sigma$ then $\bullet \vdash F(m) \leq \sigma$.*
2. *If $\bullet \vdash F(m) \leq \sigma$ then $\exists m'.\ m \rightarrow m'$ and $\bullet \vdash F(m') \leq \sigma$ by a smaller derivation.*

Proof: By the givens and Lemma 10.9.9, we must have that $\rho_1 = \lambda\alpha_1, \ldots, \alpha_n.\, m''$ for some $\alpha_1, \ldots, \alpha_n$, and $m'' \Rightarrow m \rightarrow m'$ where $m' = [\rho_n/\alpha_n] \ldots [\rho_1/\alpha_1]m''$ (ROW). We can then reason as follows with $|\{\beta, \alpha'_1, \ldots, \alpha'_n, \alpha''_1, \ldots, \alpha''_n\}| = 2n+1$:
$\bullet \vdash \mathrm{F}(m) \leq \sigma \Leftrightarrow \bullet \vdash \mathrm{F}(<\rho_1, \ldots, \rho_n>) \leq \sigma$
$\Leftrightarrow \bullet \vdash \exists\beta{=}\sigma.\exists\alpha'_1{=}\mathrm{F}(\rho_1). \cdots \exists\alpha'_n{=}\mathrm{F}(\rho_n).\neg\alpha'_1 \leq$
$\exists\beta.\exists\alpha'_1. \cdots \exists\alpha'_n.\neg\exists\alpha''_1{=}\alpha'_1. \cdots \exists\alpha''_n{=}\alpha'_n.\neg\beta$
$\Leftrightarrow \bullet \vdash \exists\alpha'_1{=}\mathrm{F}(\rho_1). \cdots \exists\alpha'_n{=}\mathrm{F}(\rho_n).\neg\alpha'_1 \leq \exists\alpha'_1. \cdots \exists\alpha'_n.\neg\exists\alpha''_1{=}\alpha'_1. \cdots \exists\alpha''_n{=}\alpha'_n.\neg\sigma$
(Lemma 10.9.12, 10.9.13, SUM-O, SU-FOR, and the given)
$\Leftrightarrow \bullet \vdash \neg\mathrm{F}(\rho_1) \leq \neg\exists\alpha''_1{=}\mathrm{F}(\rho_1). \cdots \exists\alpha''_n{=}\mathrm{F}(\rho_n).\neg\sigma$ (Lemma 10.9.12, 10.9.13, SUM-O, SU-FOR, and the given)
$\Leftrightarrow \bullet \vdash \exists\alpha''_1{=}\mathrm{F}(\rho_1). \cdots \exists\alpha''_n{=}\mathrm{F}(\rho_n).\neg\sigma \leq \mathrm{F}(\rho_1)$ (SU-NEG)
$\Leftrightarrow \bullet \vdash \exists\alpha''_1{=}\mathrm{F}(\rho_1). \cdots \exists\alpha''_n{=}\mathrm{F}(\rho_n).\neg\sigma \leq \mathrm{F}(\lambda\alpha_1, \ldots, \alpha_n.\, m'')$
$\Leftrightarrow \bullet \vdash \exists\alpha''_1{=}\mathrm{F}(\rho_1). \cdots \exists\alpha''_n{=}\mathrm{F}(\rho_n).\neg\sigma \leq \exists\alpha_1. \cdots \exists\alpha_n.\neg\mathrm{F}(m'')$
$\Leftrightarrow \bullet \vdash \neg\sigma \leq \neg[\mathrm{F}(\rho_n)/\alpha_n] \ldots [\mathrm{F}(\rho_1)/\alpha_1]\mathrm{F}(m'')$ (Lemma 10.9.12, 10.9.13, SUM-O, SU-FOR, and the given)
$\Leftrightarrow \bullet \vdash [\mathrm{F}(\rho_n)/\alpha_n] \ldots [\mathrm{F}(\rho_1)/\alpha_1]\mathrm{F}(m'') \leq \sigma$ (SU-NEG)
$\Leftrightarrow \bullet \vdash \mathrm{F}([\rho_n/\alpha_n] \ldots [\rho_1/\alpha_1]m'') \leq \sigma$ (Lemma 10.9.12)
$\Leftrightarrow \bullet \vdash \mathrm{F}(m') \leq \sigma$. □

Theorem [Rowing] 10.9.16 *Suppose m is a closed n-width row machine. Then m halts iff $\bullet \vdash F(m) \leq \sigma$.*

Proof: The forward direction follows from Lemma 10.9.14, part one of Lemma 10.9.15, and Lemma 10.9.7. The backward direction follows from part two of Lemma 10.9.15 and Lemma 10.9.7. □

The desired undecidability results then follow immediately:

Theorem [Simple] 10.9.17 *Simple subtyping is undecidable.*

Proof: Follows immediately from Theorem 10.9.16 and Theorem 10.9.10. □

Corollary 10.9.18 *The kernel-system subtyping problem is undecidable.*

Proof: Follows immediately from Theorem 10.9.17 and Corollary 10.8.15. □

Is the undecidability of subtyping likely to be a problem in practice? I don't think there is much chance of it being a problem with human generated code. Consider, for example, the previous looping example, which is the smallest such example known. Examining it reveals that a type at least as complicated as $\neg G_{\alpha}(P(\alpha)) = \neg\exists\alpha.\neg\exists\alpha'{=}\alpha.\neg\alpha$ is needed to start the loop going.

Translated into SML-like code, using functors to implement negations, this type would look like the following:

```
FUNCTOR (s: interface
              type T;
              FUNCTOR F(s: interface
                             type U = T;
                             FUNCTOR G(s: U):RES;
                           end):RES;
            end):RES
```

where RES is any type. The translated type is quite complicated, involving a functor that takes an argument containing a functor that takes an argument containing yet another functor. Types of this sort seem unlikely in practice to arise in human generated code.

Machine generated code is a different matter: it seems plausible that some kind of automatic encoding of objects, for example, might produce types of this sort. I suggest that this problem be handled by adding a depth limit to the subtyping procedure so that it returns an error if it cannot resolve the subtyping question within the specified depth limit. An initial limit of ten thousand should allow all reasonable programs to be type checked without problem (I shall show in Chapter 12 that well-typed programs never produce looping) ; if the programmer is unsure if an error is due to looping or to the limit, she can increase the limit to a much higher value and recheck the program. If the program still gives an error, either it really does have a type error, or, alternatively, it just takes so long to type check that it is unusable in practice.

Chapter 11

Soundness

I have now finished discussing the kernel-system kind and constructor levels and their associated proofs. In this chapter, I introduce the term level, define the system's semantics, and prove soundness. I defer the discussion of how to efficiently type check terms to the next chapter.

11.1 The Terms

The syntax for the term level is as follows:

Definition 11.1.1 (Syntax for the term level)

$$\begin{array}{lll} \textit{Terms} & M & ::= & x \mid \lambda x{:}A.\, M \mid M_1\, M_2 \mid (M_1,\, M_2) \mid M.1 \mid M.2 \mid {<}A{>} \mid \\ & & & M{<:}A \mid \textbf{roll} \mid \textbf{unroll} \mid \textbf{new} \mid \textbf{get} \mid \textbf{set} \end{array}$$

Scoping is as would be expected and the definitions of free constructor variables (FCV(M)) and free term variables (FTV(M)) on terms are the obvious ones:

Definition 11.1.2 (Free constructor variables)

$$\begin{array}{rcl}
FCV(x) & = & \emptyset \\
FCV(\mathbf{roll}) & = & \emptyset \\
FCV(\mathbf{unroll}) & = & \emptyset \\
FCV(\mathbf{new}) & = & \emptyset \\
FCV(\mathbf{get}) & = & \emptyset \\
FCV(\mathbf{set}) & = & \emptyset \\
\\
FCV(M.1) & = & FCV(M) \\
FCV(M.2) & = & FCV(M) \\
FCV(<A>) & = & FCV(A) \\
\\
FCV(\lambda x{:}A.\, M) & = & FCV(A) \cup FCV(M) \\
FCV(M{<:}A) & = & FCV(M) \cup FCV(A) \\
FCV(M_1\, M_2) & = & FCV(M_1) \cup FCV(M_2) \\
FCV((M_1,\, M_2)) & = & FCV(M_1) \cup FCV(M_2)
\end{array}$$

Definition 11.1.3 (Free term variables)

$$\begin{array}{rcl}
FTV(x) & = & \{x\} \\
FTV(\lambda x{:}A.\, M) & = & FTV(A) \cup (FTV(M) \Leftrightarrow \{x\}) \\
\\
FTV(\mathbf{roll}) & = & \emptyset \\
FTV(\mathbf{unroll}) & = & \emptyset \\
FTV(\mathbf{new}) & = & \emptyset \\
FTV(\mathbf{get}) & = & \emptyset \\
FTV(\mathbf{set}) & = & \emptyset \\
\\
FTV(M.1) & = & FTV(M) \\
FTV(M.2) & = & FTV(M) \\
FTV(<A>) & = & FTV(A) \\
\\
FTV(M_1\, M_2) & = & FTV(M_1) \cup FTV(M_2) \\
FTV((M_1,\, M_2)) & = & FTV(M_1) \cup FTV(M_2) \\
FTV(M{<:}A) & = & FTV(M) \cup FTV(A)
\end{array}$$

The term level's only judgment is the *well-typed term* judgment $(, \vdash M : A)$ which assigns a type (A) to a term (M) that is well-typed under a assignment (,). When discussing the types of terms, the following abbreviations will be useful:

Definition 11.1.4 (Arrow types)
The arrow type $A{\to}A'$ is defined to be equal to $\Pi x{:}A.\, A'$ where $x \notin FTV(A')$. The arrow operator ($\to$) is defined to have lower precedence than application ($A_1 A_2$).

Definition 11.1.5 (Pair types)
The pair type (A, A') is defined to be equal to $\Sigma x{:}A.\, A'$ where $x \notin FTV(A')$.

Definition 11.1.6 (Polymorphic types)
The polymorphic type $\forall\alpha{::}K.A$ is defined to be equal to $\Pi x{:}{<}K{>}.\, [x!/\alpha]A$ where $x \notin FTV(A)$.

The rules for the well-typed term judgment follow. The elimination rules for dependent functions (T-APP) and dependent sums (T-SND) are restricted so that they apply only to terms with non-dependent types. The introduction rule for reified constructors (T-REIFY) always assigns a transparent type; an opaque type can be obtained instead by using subsumption afterwards (i.e., use T-REIFY followed by T-SUMP). The rule for term variables (T-VAR) uses the *self* function ($[\mathrm{self}{=}x\rho]\Leftrightarrow$), discussed below, to get the effect of a series of EVALUE rules in a row; the advantage of the self function, which is used only at term-variable introduction time, over separate EVALUE rules, which can be applied at any time, is that it results in normalized typing derivations, which are easier to reason about. I discussed how these kind of rules work together to allow information about constructor components to be propagated solely through the constructor level in Chapter 4 and Section 5.5.

Definition 11.1.7 (Term Formation Rules)

$$\frac{,\ \vdash\ ,\ (x) = A :: \Omega}{,\ \vdash x : [\mathit{self}{=}x]A} \qquad \text{(T-VAR)}$$

$$\frac{,\ , x{:}A \vdash M : A'}{,\ \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, A'} \qquad \text{(T-LAM)}$$

$$\frac{,\ \vdash M_1 : A_2{\to}A \qquad ,\ \vdash M_2 : A_2}{,\ \vdash M_1\, M_2 : A} \qquad \text{(T-APP)}$$

$$\frac{,\ \vdash M_1 : A_1 \qquad ,\ \vdash M_2 : A_2}{,\ \vdash (M_1,\, M_2) : (A_1, A_2)} \qquad \text{(T-PAIR)}$$

$$\frac{,\ \vdash M : \Sigma x{:}A_1.\, A_2}{,\ \vdash M.1 : A_1} \qquad \text{(T-FST)}$$

$$\frac{,\ \vdash M : (A_1, A_2)}{,\ \vdash M.2 : A_2} \qquad \text{(T-SND)}$$

$$\frac{,\ \vdash A :: K}{,\ \vdash \,<\!A\!>\, : \,<\!=\!A\!::\!K\!>} \qquad \text{(T-REIFY)}$$

$$\frac{,\ \vdash M : A' \qquad ,\ \vdash A' \leq A}{,\ \vdash M : A} \qquad \text{(T-SUMP)}$$

$$\frac{,\ \vdash M : A}{,\ \vdash M{<:}A : A} \qquad \text{(T-COERCE)}$$

$$\frac{\vdash\ ,\ \ \textit{valid}}{,\ \vdash \mathbf{new} : \forall\alpha{::}\Omega.\alpha{\to}\mathbf{ref}\,\alpha} \qquad \text{(T-NEW)}$$

$$\frac{\vdash\ ,\ \ \textit{valid}}{,\ \vdash \mathbf{get} : \forall\alpha{::}\Omega.\mathbf{ref}\,\alpha{\to}\alpha} \qquad \text{(T-GET)}$$

$$\frac{\vdash\ ,\ \ \textit{valid}}{,\ \vdash \mathbf{set} : \forall\alpha{::}\Omega.\mathbf{ref}\,\alpha{\to}(\alpha{\to}\alpha)} \qquad \text{(T-SET)}$$

$$\frac{\vdash\ ,\ \ \textit{valid}}{,\ \vdash \mathbf{roll} : \forall\alpha{::}\Omega{\Rightarrow}\Omega.\alpha\,(\mathbf{rec}\,\alpha){\to}\mathbf{rec}\,\alpha} \qquad \text{(T-ROLL)}$$

$$\frac{\vdash\ ,\ \ \textit{valid}}{,\ \vdash \mathbf{unroll} : \forall\alpha{::}\Omega{\Rightarrow}\Omega.\mathbf{rec}\,\alpha{\to}\alpha\,(\mathbf{rec}\,\alpha)} \qquad \text{(T-UNROLL)}$$

The self function is defined below. Given a term $x\rho$ with type A, [self=$x\rho$]A computes a *subtype* of A for $x\rho$ by adding information about the identity of constructor components to opaque types ($<\!K\!>$) in A. For example,
[self=$x\rho$]$\Sigma x'{:}<\!=\!\alpha{::}\Omega\!>.<\!\Omega\!> = \Sigma x'{:}<\!=\!\alpha{::}\Omega\!>.<\!=\!x\rho.2!{::}\Omega\!>$. Note that the self function leaves transparent types ($<\!=\!A{::}K\!>$) unchanged because more useful information about the identity of their components is already available.

Definition 11.1.8 (Self function)

$$\begin{array}{rcll}
[\mathit{self}{=}x\rho]{<}K{>} & = & {<}{=}x\rho!{::}K{>} & \\
[\mathit{self}{=}x\rho]\Sigma x'{:}A_1.\,A_2 & = & \Sigma x'{:}[\mathit{self}{=}x\rho.1]A_1.\,[\mathit{self}{=}x\rho.2]A_2, & \textit{where } x \neq x' \\
 & & & \\
[\mathit{self}{=}x\rho]\alpha & = & \alpha & \\
[\mathit{self}{=}x\rho]\Pi x'{:}A_1.\,A_2 & = & \Pi x'{:}A_1.\,A_2 & \\
[\mathit{self}{=}x\rho]\lambda\alpha{::}K.\,A & = & \lambda\alpha{::}K.\,A & \\
[\mathit{self}{=}x\rho]A_1\,A_2 & = & A_1\,A_2 & \\
[\mathit{self}{=}x\rho]{<}{=}A{::}K{>} & = & {<}{=}A{::}K{>} & \\
[\mathit{self}{=}x\rho]x'\rho'! & = & x'\rho'! & \\
[\mathit{self}{=}x\rho]\mathbf{rec} & = & \mathbf{rec} & \\
[\mathit{self}{=}x\rho]\mathbf{ref} & = & \mathbf{ref} &
\end{array}$$

The self function does not take into account equality on types; it leaves unchanged types with shape ?, regardless of their intrinsic shape. For example, [self=x]$((\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>})$ = $(\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>}$ even though [self=x]${<}\Omega{>}$ = ${<}{=}x!{::}\Omega{>}$ and
$\bullet \vdash (\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>} = {<}\Omega{>} :: \Omega$.

The effect of any series of applications of EVALUE-like rules can be gotten by first using equality to rewrite the type so that only the opaque types you want to alter have non-? shapes and then applying the self function. For instance, suppose $\vdash$, valid and , $(x) = \Sigma x'{:}{<}\Omega{>}.\,{<}\Omega{>}$ and we want to add information to only the second component of x. By the equality rules, we have that , $\vdash$, $(x) = \Sigma x'{:}(\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>}.\,{<}\Omega{>} :: \Omega$. By the definition of the self function, [self=x]$\Sigma x'{:}(\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>}.\,{<}\Omega{>}$ =
$\Sigma x'{:}(\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>}.\,{<}{=}x.2!{::}\Omega{>}$. Hence, by T-VAR, we have
, $\vdash x : \Sigma x'{:}(\lambda\alpha{::}\Omega.\,\alpha)\,{<}\Omega{>}.\,{<}{=}x.2!{::}\Omega{>} \Rightarrow$, $\vdash x : \Sigma x'{:}{<}\Omega{>}.\,{<}{=}x.2!{::}\Omega{>}$ (T-SUMP, etc.) as desired.

11.2 Self Validity

In this section I prove that under suitable conditions the self function produces valid types. In order to describe when the self function is valid, a notion of *cleaving* is needed. The cleave function (, ,$A \Downarrow \rho$), defined below, takes as input an assignment , , a type A under , , and a path ρ. Treating A like a tree whose branches are named by .1 and .2, it cleaves apart A following the path ρ into three parts: the part of A rooted at ρ, call it A'; the declarations in A that A' is under (the tree to the left of ρ), call it , $'$; and the remaining part (the tree to the right of ρ). The cleave function then returns , ; , $'$, A', discarding the right part of A.

Definition 11.2.1 (Cleave function)

$$
\begin{array}{rcl}
,\ ,A \Downarrow \epsilon & = & ,\ ,A \\
\\
,\ ,\Sigma x'{:}A_1.\,A_2 \Downarrow .1\rho & = & ,\ ,A_1 \Downarrow \rho \\
,\ ,\Sigma x'{:}A_1.\,A_2 \Downarrow .2\rho & = & (,\ ,x'{:}A_1),A_2 \Downarrow \rho \quad \textit{where } x' \notin \textit{dom}(,\ )
\end{array}
$$

Like the selection function, the cleave function can also be done in steps:

Lemma 11.2.2 *If* $,\ ,A \Downarrow \rho_1\rho_2$ *exists then* $,\ ,A \Downarrow \rho_1\rho_2 = ,\ _2, A_2$ *where* $,\ _2, A_2 = ,\ _1,A_1 \Downarrow \rho_2$ *and* $,\ _1, A_1 = ,\ ,A \Downarrow \rho_1$.

Given a valid assignment, type pair, cleaving results in a new valid assignment, type pair:

Lemma 11.2.3 (Cleave validity)
If $,\ \vdash A :: \Omega$ *and* $,\ ', A' = ,\ ,A \Downarrow \rho$ *then* $,\ ' \vdash A' :: \Omega$.

The types produced by either cleaving or selecting from the same type A using the same path are related via a series of place substitutions on the term variables bound in A:

Lemma 11.2.4 (Shape preservation) $\lceil [x\rho/x']A \rceil = \lceil A \rceil$

Lemma 11.2.5 (Cleaving vs. selection)
If $,\ ,A \Downarrow \rho'$ *or* $\mathcal{S}(A, x\rho, \rho')$ *exist, then* $\exists$ *a series of place substitutions,* $\theta = \theta_1 \cdots \theta_n$*, such that* $\mathcal{S}(A, x\rho, \rho') = \theta A'$ *and* $\textit{dom}(\theta) = \textit{dom}(,\ ') \Leftrightarrow \textit{dom}(,\ )$ *where* $,\ ', A' = ,\ ,A \Downarrow \rho'$.

Proof: Proved by structural induction on ρ'. Cases:

$\rho' = \epsilon$: $\Rightarrow ,\ ' = ,\ , A' = A$, and $\mathcal{S}(A, x\rho, \rho') = A \Rightarrow \mathcal{S}(A, x\rho, \rho') = A' = \theta A'$ and $\text{dom}(\theta) = \emptyset = \text{dom}(,\ ') \Leftrightarrow \text{dom}(,\ )$ if $n = 0$.

$\rho' = \rho''.i$: $\Rightarrow ,\ ,A \Downarrow \rho''$ or $\mathcal{S}(A, x\rho, \rho'')$ exist (Lemmas 6.5.3 and 11.2.2). Hence, by the induction hypothesis, $\exists$ a series of place substitutions, $\theta = \theta_1 \cdots \theta_n$, such that $\mathcal{S}(A, x\rho, \rho'') = \theta A''$ and $\text{dom}(\theta) = \text{dom}(,\ '') \Leftrightarrow \text{dom}(,\ )$ where $,\ '', A'' = ,\ ,A \Downarrow \rho'' \Rightarrow \mathcal{S}((\theta A''), x\rho\rho'', .i)$ or $,\ '',A'' \Downarrow .i$ exist (Lemmas 6.5.3 and 11.2.2) $\Rightarrow \lceil \theta A'' \rceil = \lceil A'' \rceil = \Sigma$ (Lemma 11.2.4) $\Rightarrow \exists x', A_1, A_2.\ A'' = \Sigma x'{:}A_1.\,A_2$ where $x \neq x'$ and $x' \notin \text{dom}(,\ )$ $\Rightarrow \theta A'' = \Sigma x'{:}\theta A_1.\,\theta A_2$ (WLOG) $\Rightarrow ,\ ' = ,\ '', A' = A_i$, and $\mathcal{S}(A, x\rho, \rho') = \theta A_1$ (if $i = 1$) or $,\ ' = ,\ '', x'{:}A_1$ and $\mathcal{S}(A, x\rho, \rho') = [x\rho\rho''.1/x'](\theta A_2)$ (if $i = 2$) (Lemmas 6.5.3 and 11.2.2) $\Rightarrow \exists \theta'.\ \mathcal{S}(A, x\rho', \rho') = \theta' A'$ and $\text{dom}(\theta') = \text{dom}(,\ ') \Leftrightarrow \text{dom}(,\ )$.

□

One corollary of this result is that such types must have the same shape:

Corollary 11.2.6 *If $, ,A \Downarrow \rho'$ or $\mathcal{S}(A, x\rho, \rho')$ exist, then $\lceil \mathcal{S}(A, x'\rho', \rho)\rceil = \lceil A' \rceil$ where $, ', A' = , ,A \Downarrow \rho$*

Proof: Follows directly from Lemmas 11.2.5 and 11.2.4. □

Another corollary is that if the part of A being selected or cleaved to is not bound by any term variables declared in A, then the type obtained by selection and cleaving will be the same:

Lemma 11.2.7 (Substitution for non-free variables)

1. *If $\alpha \notin FCV(A)$ then $[A'/\alpha]A = A$.*
2. *If $x' \notin FTV(A)$ then $[x\rho/x']A = A$.*
3. *If $\alpha \notin FCV(,)$ then $[A'/\alpha], = , $.*
4. *If $x' \notin FTV(,)$ then $[x\rho/x'], = , $.*

Proof: Proved sequentially using structural induction on A and $, $. □

Corollary 11.2.8 *If $, \vdash A :: \Omega$, $, ', A' = , ,A \Downarrow \rho$, and $, \vdash A' :: \Omega$ then $\mathcal{S}(A, x\rho', \rho) = A'$.*

Proof: By Lemma 11.2.3, $, ' \vdash A' :: \Omega$. By Lemma 11.2.5, $\exists$ a series of place substitutions, $\theta = \theta_1 \cdots \theta_n$, such that $\mathcal{S}(A, x\rho', \rho) = \theta A'$ and $\text{dom}(\theta) = \text{dom}(, ') \Leftrightarrow \text{dom}(,)$. By inspection of the definition of cleaving, $, ' = , ; , ''$ for some $, ''$. By Lemmas 6.3.5, 6.5.10, and 6.5.4, $\text{dom}(,) \cap \text{dom}(, '') = \emptyset \Rightarrow \text{dom}(, ') \Leftrightarrow \text{dom}(,) = \text{dom}(, '')$. By Theorem 6.5.6, $\text{FTV}(A') \subseteq \text{dom}(,) \Rightarrow \text{FTV}(A') \cap \text{dom}(, '') = \emptyset \Rightarrow \text{FTV}(A') \cap \text{dom}(\theta) = \emptyset \Rightarrow \theta A' = A'$ (Lemma 11.2.7) $\Rightarrow \mathcal{S}(A, x\rho', \rho) = A'$. □

If we ignore the question of when the self function produces valid types, it is straightforward to show that the self function produces a subtype of the type it is applied to:

Lemma 11.2.9 *If $, \vdash A :: \Omega$ and $, \vdash [self{=}x\rho]A :: \Omega$ then $, \vdash [self{=}x\rho]A \leq A$.*

Proof: Proved by structural induction on A. All but the following two cases follow immediately from O-REFL and S-ONE:

Opaq: Here $A = <K>$ for some $K \Rightarrow [\text{self}{=}x\rho]A = <=x\rho!::K>$
$\Rightarrow ,\ \vdash [\text{self}{=}x\rho]A \leq A$ (C-TRANS, O-FORGET, and S-ONE).

DSum: Here $A = \Sigma x'{:}A_1.\, A_2$ for some x', A_1, and A_2 where $x \neq x'$ and $x' \notin \text{dom}(,)$
$\Rightarrow [\text{self}{=}x\rho]A = \Sigma x'{:}[\text{self}{=}x\rho.1]A_1.\, [\text{self}{=}x\rho.2]A_2 \Rightarrow ,\ \vdash A_1 :: \Omega,\ ,\ , x'{:}A_1 \vdash A_2 :: \Omega,$
$,\ \vdash [\text{self}{=}x\rho.1]A_1 :: \Omega$, and $,\ , x'{:}[\text{self}{=}x\rho.1]A_1 \vdash [\text{self}{=}x\rho.2]A_2 :: \Omega$ (C-DSUM, Lemma 6.3.5, and DECL-T) $\Rightarrow ,\ \vdash [\text{self}{=}x\rho.1]A_1 \leq A_1$ and
$,\ , x'{:}[\text{self}{=}x\rho.1]A_1 \vdash [\text{self}{=}x\rho.2]A_2 \leq A_2$ (induction hypothesis) $\Rightarrow$
$,\ \vdash [\text{self}{=}x\rho]A \leq A$. (Theorem 10.5.4)

□

Using these results, I can now show when the self function produces valid types. The cleave function is needed here because the self function recurses in a cleave-like manner while, on the other hand, the place lookup judgment, which is needed to ensure the validity of the inserted names, recurses in a selection-like manner.

Theorem 11.2.10 (Self validity)
If $,\ \vdash x\rho \Rightarrow A$ *and* $,', A' = ,, A \Downarrow \rho'$ *then* $,' \vdash [\textit{self}{=}x\rho\rho']A' :: \Omega$.

Proof: Proved by structural induction on A'. Theorem 9.4.4 and Lemma 11.2.3 handle the cases where $\lceil A' \rceil \neq \Sigma$ and $\lceil A' \rceil \neq <K>$. The other cases are:

Opaq: Given $A' = <K>$ for some $K \Rightarrow [\text{self}{=}x\rho\rho']A' = <=x\rho\rho'!::K>$. By Lemma 11.2.3,
$,' \vdash A' :: \Omega \Rightarrow \vdash ,'$ valid (Theorem 6.3.5). By Theorem 9.4.4 and E-REFL,
$,\ \vdash x\rho\rho' \Rightarrow \mathcal{S}(A', x\rho, \rho') \Rightarrow ,\ \vdash x\rho\rho' \Rightarrow <K>$ (Corollary 11.2.6) $\Rightarrow ,' \vdash x\rho\rho' \Rightarrow <K>$
(Theorem 9.4.1) $\Rightarrow ,' \vdash x\rho\rho'! :: K$ (C-EXT-O) $\Rightarrow ,' \vdash <=x\rho\rho'!::K> :: \Omega$ (C-TRANS)
$\Rightarrow ,' \vdash [\text{self}{=}x\rho\rho']A' :: \Omega$.

DSum: Given $A' = \Sigma x'{:}A_1.\, A_2$ for some x', A_1, and A_2 where $x \neq x'$ and $x' \notin \text{dom}(,')$.
By Theorem 9.4.4 and Lemma 11.2.3, $,' \vdash A_1 :: \Omega$. By the induction hypothesis,
$,' \vdash [\text{self}{=}x\rho\rho'.1]A_1 :: \Omega$ and $,', x'{:}A_1 \vdash [\text{self}{=}x\rho\rho'.2]A_2 :: \Omega$.
$\Rightarrow ,' \vdash [\text{self}{=}x\rho.1]A_1 \leq A_1$. (Lemma 11.2.9)
$\Rightarrow ,', x'{:}[\text{self}{=}x\rho.1]A_1 \vdash [\text{self}{=}x\rho\rho'.2]A_2 :: \Omega$ (Theorem 10.4.7)
$\Rightarrow ,' \vdash \Sigma x'{:}[\text{self}{=}x\rho\rho'.1]A_1.\, [\text{self}{=}x\rho\rho'.2]A_2 :: \Omega$ (C-DSUM)
$\Rightarrow ,' \vdash [\text{self}{=}x\rho\rho']A' :: \Omega$.

□

Corollary 11.2.11 *If* $, \vdash x\rho \Rightarrow A$ *and* $,', A' = , ,A \Downarrow \rho'$ *then* $,' \vdash [\mathit{self}{=}x\rho\rho']A' \leq A'$.

Proof: By Theorem 9.4.4 and Lemma 11.2.3, $,' \vdash A' :: \Omega$. By Theorem 11.2.10, $,' \vdash [\mathrm{self}{=}x\rho\rho']A' :: \Omega$. Hence, by Lemma 11.2.9, $,' \vdash [\mathrm{self}{=}x\rho\rho']A' \leq A'$. □

The following lemma, which describes the interaction between subtyping and cleaving, will be useful in the next section:

Lemma 11.2.12 *Suppose* $, \vdash A_1 \leq A_2$, $,''_1, A''_1 = , ,A_1 \Downarrow \rho$, *and* $,''_2, A''_2 = , ,A_2 \Downarrow \rho$. *Then:*

1. $,''_1 \vdash A''_1 \leq A''_2$
2. *If* $,''_2; ,_3 \vdash A :: K$ *then* $,''_1; ,_3 \vdash A :: K$.
3. *If* $,''_2; ,_3 \vdash A \leq A'$ *then* $,''_1; ,_3 \vdash A \leq A'$.

Proof: Proved simultaneously by structural induction on ρ:

$\rho = \epsilon$: Here $, = ,''_1 = ,''_2$ so we are done.

$\rho = \rho'.1$: Here $\exists A_4, A_5, x'.\ , ,A_1 \Downarrow \rho' = ,'_1, \Sigma x'{:}A''_1.\, A_4$ and $, ,A_2 \Downarrow \rho' = ,'_2, \Sigma x'{:}A''_2.\, A_5$, $,''_1 = ,'_1$, and $,''_2 = ,'_2$, where $x' \notin \mathrm{dom}(,'_1) \cup \mathrm{dom}(,'_2) \cup \{x\}$. By the induction hypothesis, $,'_1 \vdash \Sigma x'{:}A''_1.\, A_4 \leq \Sigma x'{:}A''_2.\, A_5 \Rightarrow ,'_1 \vdash A''_1 \leq A''_2$ and $,'_1, x'{:}A''_1 \vdash A_4 \leq A_5$ (Lemma 10.5.1) $\Rightarrow ,''_1 \vdash A''_1 \leq A''_2$. Parts two and three follow from the induction hypothesis.

$\rho = \rho'.2$: Here $\exists A_4, A_5, x'.\ , ,A_1 \Downarrow \rho' = ,'_1, \Sigma x'{:}A_4.\, A''_1$ and $, ,A_2 \Downarrow \rho' = ,'_2, \Sigma x'{:}A_5.\, A''_2$, $,''_1 = ,'_1, x'{:}A_4$, and $,''_2 = ,'_2, x'{:}A_5$, where $x' \notin \mathrm{dom}(,'_1) \cup \mathrm{dom}(,'_2) \cup \{x\}$. By the induction hypothesis, $,'_1 \vdash \Sigma x'{:}A_4.\, A''_1 \leq \Sigma x'{:}A_5.\, A''_2 \Rightarrow ,'_1 \vdash A_4 \leq A_5$ and $,'_1, x'{:}A_4 \vdash A''_1 \leq A''_2$ (Lemma 10.5.1) $\Rightarrow ,''_1 \vdash A''_1 \leq A''_2$. Parts two and three follow from the induction hypothesis followed by replacement by a subtype (Theorem 10.4.7).

□

11.3 Self and Subtyping

In this section I establish results about the interaction between the self function and the subtyping relation that are needed to extended the replacement by a subtype result to the term level. First, I shall need some minor lemmas relating self to selection and to cleaving:

Lemma 11.3.1 *If $x \neq x'$ then $[self{=}x\rho][x''\rho'/x']A = [x''\rho'/x'][self{=}x\rho]A$.*

Lemma 11.3.2 (Self and selection)
If either $\mathcal{S}([self{=}x_2\rho_2]A, x_1\rho_1, \rho)$ or $\mathcal{S}(A, x_1\rho_1, \rho)$ exists, then $\mathcal{S}([self{=}x_2\rho_2]A, x_1\rho_1, \rho) = [self{=}x_2\rho_2\rho]\mathcal{S}(A, x_1\rho_1, \rho)$.

Proof: Proved by structural induction on ρ using Lemma 11.3.1 as needed. Example case:

$$\begin{array}{l}
\mathcal{S}([\mathrm{self}{=}x_2\rho_2]\Sigma x'{:}A_1.\, A_2, x_1\rho_1, .2\rho') = \\
\mathcal{S}(\Sigma x'{:}[\mathrm{self}{=}x_2\rho_2.1]A_1.\, [\mathrm{self}{=}x_2\rho_2.2]A_2, x_1\rho_1, .2\rho') = \\
\mathcal{S}([x_2\rho_2.1/x'][\mathrm{self}{=}x_2\rho_2.2]A_2, x_1\rho_1.2, \rho') = \\
\mathcal{S}([\mathrm{self}{=}x_2\rho_2.2][x_2\rho_2.1/x']A_2, x_1\rho_1.2, \rho') = \\
[\mathrm{self}{=}x_2\rho_2.2\rho']\mathcal{S}([x_1\rho_1.1/x']A_2, x_1\rho_1.2, \rho') = \\
[\mathrm{self}{=}x_2\rho_2.2\rho']\mathcal{S}(\Sigma x'{:}A_1.\, A_2, x_1\rho_1, .2\rho')
\end{array}$$

(x' chosen so that $x' \neq x_2$) □

Lemma 11.3.3 (Self and cleaving)
If $,', A' = , ,A \Downarrow \rho$ then $\exists ,''. \; , ,[self{=}x\rho']A \Downarrow \rho = , '', [self{=}x\rho'\rho]A'$.

Proof: Proved by structural induction on ρ. □

I shall also need the following result which shows that an equality on part of a type can be extended to an equality on the entire type:

Lemma 11.3.4 (Equality back off)
Suppose $, \vdash A_1 :: \Omega$, $,_1, A_1' = , ,A_1 \Downarrow \rho'$, and $,_1 \vdash A_1' = A_2' :: \Omega$. Then, $\exists ,_2, A_2$ such that:

1. $,_2, A_2' = , ,A_2 \Downarrow \rho'$
2. $, \vdash A_1 = A_2 :: \Omega$

Proof: Proved by structural induction on ρ'. Cases:

$\rho' = \epsilon$: $\Rightarrow ,_1 = ,$ and $A_1 = A_1'$. Let $,_2 = ,$ and $A_2 = A_2'$ and we are done.

$\rho' = \rho''.i$: Here $,_1', \Sigma x'{:}A_3.\, A_4 = , ,A_1 \Downarrow \rho''$ for some $,_1'$, x', A_3, and A_4 where $x' \neq x$ and $x' \notin \mathrm{dom}(,_1')$ (Lemma 11.2.2) $\Rightarrow A_1' = A_{i+2}$. By Lemma 11.2.3, $,_1' \vdash A_3 :: \Omega$ and $,_1', x'{:}A_3 \vdash A_4 :: \Omega \Rightarrow ,_1' \vdash A_3 = A_3 :: \Omega$ and $,_1', x'{:}A_3 \vdash A_4 = A_4 :: \Omega$ (E-REFL) $\Rightarrow ,_1' \vdash \Sigma x'{:}A_3.\, A_4 = A_5 :: \Omega$ where $A_5 = \Sigma x'{:}A_2'.\, A_4$ if $i = 1$ and $A_5 = \Sigma x{:}A_3.\, A_2'$ if $i = 2$.

Hence, by the induction hypothesis, $\exists ,_0, A_2. \; ,_0, A_5 = , ,A_2 \Downarrow \rho''$ and $, \vdash A_1 = A_2 :: \Omega$ $\Rightarrow \exists ,_2. \; , ,A_2 \Downarrow \rho''.i = ,_2, A_2'$.

□

The first key result I need is that if x is bound to a semi-canonical type A in , (i.e., , $=$, $_1, x{:}A;$, $_2$ and $A \in C_{,\,1}$), then the result of selecting using path ρ on A ($\mathcal{S}(A, x, \rho)$) can also be obtained by means of equality on the type resulting from cleaving A with ρ under the assignment resulting from cleaving , , [self=x]A with ρ. (See Theorem 11.3.9 below for the formal version of this result.)

In essence, this result says that applying the self function to the declarations binding a subcomponent of A adds enough information to allow converting back and forth between the internal and external names for the preceding subcomponents. (Selection converts internal names to external ones; cleaving leaves names unchanged.) For example, if $A = \Sigma x'{:}{<}\Omega{>}.\, x'!$ and $\rho = .2$, then we will have the following:

$$, , x'{:}[\text{self}{=}x.1]{<}\Omega{>} \vdash x'! = x.1! :: \Omega$$

The restriction that A be a semi-canonical type is needed here to ensure that sufficient information is added by self. (Because semi-canonical types have maximally defined shapes, self applied to them gives the maximal amount of information.) If we gave x' the non-semi-canonical type $(\lambda\alpha{::}\Omega.\,\alpha){<}\Omega{>}$ instead, the desired equation would not hold.

The proof of this result is quite complicated. First, I show that constructor extractions involving a component of A can have their names converted if we can use the final result on subcomponents of the component in question (Lemma 11.3.7). Second, I show how to extend the first lemma's result from constructor extractions to arbitrary types (Lemma 11.3.8). And finally, third, I use the previous lemmas to prove the desired result (Theorem 11.3.9).

The key idea behind the third step is to apply Lemma 11.3.7 each time we cleave off a declaration so as to convert all references to that component in the type remaining so they use the component's external name; by using the fact that self produces a subtype and the replacement by a subtype result, we can connect together the resulting series of equalities to convert all the references. The recursion involved can be shown to terminate by showing that all the recursive calls are to *lexically preceding paths* (defined below) and that only a finite number of paths are valid for any given A (i.e., $\mathcal{S}(A, x, \rho)$ exists).

Definition 11.3.5 (Lexical order for paths)
The path ρ_1 lexically precedes the path ρ_2 (written $\rho_1 < \rho_2$) iff one of the following:

1. $\rho_1 = \epsilon$ *and* $\rho_2 \neq \epsilon$
2. $\rho_1 = .1\rho_1'$ *and* $\rho_2 = .2\rho_2'$
3. $\rho_1 = .i\rho_1'$, $\rho_2 = .i\rho_2'$, *and* $\rho_1' < \rho_2'$

Lemma 11.3.6 *If* $, _1, x{:}A; , _2 \vdash x\rho! :: K$ *and* $A \in C_{, _1}$ *then* $, _1, x{:}A; , _2 \vdash x\rho \Rightarrow \mathcal{S}(A, x, \rho)$ *and* $\lceil \mathcal{S}(A, x, \rho) \rceil \leq \,<\!K\!>$.

Proof: Inspection of the typing rules reveals that the derivation of $, _1, x{:}A; , _2 \vdash x\rho! :: K$ must be by either the rule C-EXT-O or the rule C-EXT-T $\Rightarrow$ $\exists A'', K.\ , _1, x{:}A; , _2 \vdash x\rho \Rightarrow A''$ and $\lceil A'' \rceil \leq \,<\!K\!> \Rightarrow\ , _1, x{:}A; , _2 \vdash A'' = A' :: \Omega$ and $A' \in C_{, _1, x:A; , _2}$ where $A' = \mathcal{S}(A, x, \rho)$ (Lemma 9.5.9) $\Rightarrow \lceil A' \rceil = \lceil A'' \rceil$ (Lemma 9.5.7 and E-SYM) $\Rightarrow \lceil A' \rceil \leq \,<\!K\!>$. By P-INIT, P-MOVE, Lemma 6.3.5, Theorem 9.4.4, and E-REFL, $, _1, x{:}A; , _2 \vdash x\rho \Rightarrow A'$. □

Lemma 11.3.7 (Name conversion) *Suppose:*

1. $A \in C_{, _1}$
2. $, \ = \ , _1, x{:}A; , _2$
3. $, _0, A_0 = \ , ,([\mathit{self}{=}x]A) \Downarrow \rho_1$
4. $, '_0, A'_0 = \ , _0, A_0 \Downarrow \rho_2$
5. $, '_9, A'_9 = \ , ,A \Downarrow \rho_1\rho_2$
6. $, '_0 \vdash A'_9 = \mathcal{S}(A, x, \rho_1\rho_2) :: \Omega$
7. $, _0, x'{:}A_0; , '' \vdash x\rho_1\rho_2! :: K$

Then $, _0, x'{:}A_0; , '' \vdash x\rho_1\rho_2! = x'\rho_2! :: K$

Proof: By Lemma 11.2.2, we can let $, _9, A_9 = \ , ,A \Downarrow \rho_1$ and have $, '_9, A'_9 = \ , _9, A_9 \Downarrow \rho_2$. By Lemma 11.3.6, $\exists K.\ , _0, x'{:}A_0; , '' \vdash x\rho \Rightarrow \mathcal{S}(A, x, \rho_1\rho_2)$ and $\lceil \mathcal{S}(A, x, \rho_1\rho_2) \rceil \leq \,<\!K\!>$. Let $A' = \mathcal{S}(A, x, \rho_1\rho_2)$. Casing on $\lceil A' \rceil$:

$<\!K\!>$: Here $\lceil A' \rceil = \,<\!K\!> \Rightarrow A' = \,<\!K\!> \Rightarrow A'_9 = \,<\!K\!>$ (Corollary 11.2.6) $\Rightarrow \mathcal{S}(A_9, x', \rho_2) = \,<\!K\!>$ (Corollary 11.2.6) and $A_0 = [\text{self}{=}x\rho_1]A_9$ (Lemma 11.3.3) $\Rightarrow \mathcal{S}(A_0, x', \rho_2) = \mathcal{S}([\text{self}{=}x\rho_1]A_9, x', \rho_2) = [\text{self}{=}x\rho_1\rho_2]\!<\!K\!>$ (Lemma 11.3.2) $\Rightarrow \mathcal{S}(A_0, x', \rho_2) = \,<\!=\!x\rho_1\rho_2!{::}K\!>$ $\Rightarrow\ , ', x'{:}A_0; , '' \vdash x'\rho_2 \Rightarrow \,<\!=\!x\rho_1\rho_2!{::}K\!>$ (P-INIT, P-MOVE, E-REFL, and Theorem 9.4.4) $\Rightarrow\ , ', x'{:}A_0; , '' \vdash x\rho_1\rho_2! = x'\rho_2! :: K$ (C-EXT-T, E-ABBREV, E-REFL, E-SYM, Lemma 6.3.5, and Theorem 9.4.4).

$<=::K>$: Here $\lceil A' \rceil = <=::K> \Rightarrow \lceil A'_9 \rceil = <=::K>$ (Corollary 11.2.6) $\Rightarrow \exists A_2, A_3.\ A' = <=A_2::K>$ and $A'_9 = <=A_3::K> \Rightarrow\ ,\ ', x':A_0;\ ,\ '' \vdash x\rho_1\rho_2! = A_2 :: K$ (E-ABBREV, Theorem 9.4.4, and E-REFL) and $,\ '_0 \vdash A'_9 = <=A_2::K> :: \Omega \Rightarrow A'_0 = [\text{self}{=}x\rho_1\rho_2]A'_9$ (Lemma 11.3.3) $\Rightarrow A'_0 = A'_9 \Rightarrow\ ,\ '_0 \vdash A'_0 = <=A_2::K> :: \Omega$.

By Lemma 6.3.5, P-INIT, and Theorem 11.2.10, $,\ _0 \vdash A_0 :: \Omega$
$\Rightarrow \exists ,\ _8, A_8.\ ,\ _8, <=A_2::K> =\ ,\ _0, A_8 \Downarrow \rho_2$ and $,\ _0 \vdash A_0 = A_8 :: \Omega$ (Lemma 11.3.4) By Lemma 6.3.5, P-INIT, Theorem 9.4.4, E-REFL, and P-MOVE,
$,\ _0 \vdash <=A_2::K> :: \Omega \Rightarrow \mathcal{S}(A_8, x', \rho_2) = <=A_2::K>$ (Corollary 11.2.8)

By P-INIT and Lemma 6.3.5, $,\ _0, x':A_0;\ ,\ '' \vdash x' \Rightarrow A_0$

$\Rightarrow\ ,\ _0, x':A_0;\ ,\ '' \vdash x'\rho_2 \Rightarrow \mathcal{S}(A_8, x', \rho_2)$ (P-MOVE, Theorem 9.4.1, and Lemma 6.3.5) $\Rightarrow\ ,\ _0, x'\rho_2:A_0;\ ,\ '' \vdash x'\rho_2 \Rightarrow <=A_2::K>$

$\Rightarrow\ ,\ _0, x'\rho_2:A_0;\ ,\ '' \vdash x'\rho_2! = A_2 :: K$ (E-ABBREV, C-EXT-T, Theorem 9.4.4, and E-REFL).

$\Rightarrow\ ,\ _0, x':A_0;\ ,\ '' \vdash x\rho_1\rho_2! = x'\rho_2! :: K$ (E-SYM and E-TRAN).

□

Lemma 11.3.8 (Name conversion)
Suppose for all $,\ '$ *and* ρ', *if* $,\ ;\ ,\ ' \vdash x\rho\rho'! :: K$ *then* $,\ ;\ ,\ ' \vdash x\rho\rho'! = x'\rho'! :: K$.
Then if $,\ ;\ ,\ '' \vdash [x\rho/x']A :: K$ *then* $,\ ;\ ,\ '' \vdash A = [x\rho/x']A :: K$.

Proof: Proved by structural induction on the derivation of $,\ ;\ ,\ '' \vdash [x\rho/x']A :: K$. The proof is completely straightforward using E-REFL except for the following case:

C-EXT-? I: Here $,\ ;\ ,\ '' \vdash [x\rho/x']x'\rho'! :: K \Rightarrow\ ,\ ;\ ,\ '' \vdash x\rho\rho'! :: K$
$\Rightarrow\ ,\ ;\ ,\ ' \vdash x\rho\rho'! = x'\rho'! :: K$. (supposition)
$\Rightarrow\ ,\ ;\ ,\ ' \vdash x'\rho'! = [x\rho/x']x'\rho'! :: K$ (E-SYM).

□

Theorem 11.3.9 (Name conversion) *Suppose:*

1. $A \in C_{,\ 1}$
2. $,\ ', A' =\ ,\ _1, x{:}A;\ ,\ _2, A \Downarrow \rho$
3. $,\ _0, A_0 =\ ,\ _1, x{:}A;\ ,\ _2, ([\mathit{self}{=}x]A) \Downarrow \rho$

Then $,\ _0 \vdash A' = \mathcal{S}(A, x, \rho) :: \Omega$.

Proof: Proved by induction on the lexical order of ρ; recursive calls are only permitted on lexically preceding paths. Note that this measure is well-founded because there are only a finite number of paths for which $,_1, x{:}A;,_2, A \Downarrow \rho$ exists. Let $, \; = \; ,_1, x{:}A;,_2$. Casing on ρ:

$\rho = \epsilon$: $\Rightarrow \mathcal{S}(A, x, \rho) = A$, $A' = A$, and $,_0 = ,$. By Lemma 9.5.2, Lemma 6.3.5, and Theorem 9.4.1, $, \vdash A :: \Omega$. The desired result then follows via E-REFL.

$\rho = \rho'.1$: $\Rightarrow \exists x', A_2, A_2'.\; ,_0, \Sigma x'{:}A_0.\, A_2 = \;,([\text{self}{=}x]A) \Downarrow \rho'$ and $,', \Sigma x'{:}A'.\, A_2' = \;,A \Downarrow \rho'$ where $x' \neq x$ and $x' \notin \text{dom}(,'')\cup\text{dom}(,_0')$ (Lemma 11.2.2).
By Corollary 11.2.6 and Lemma 6.5.3,
$\exists A_1'', A_2''.\; \mathcal{S}(A, x, \rho') = \Sigma x{:}A_1''.\, A_2''$ and $\mathcal{S}(A, x, \rho) = A_1''$.

Applying part one of the induction hypothesis on ρ' gives
$,_0 \vdash \Sigma x'{:}A'.\, A_2' = \mathcal{S}(A, x, \rho') :: \Omega \Rightarrow \;,_0 \vdash \Sigma x'{:}A'.\, A_2' = \Sigma x{:}A_1''.\, A_2'' :: \Omega$
$\Rightarrow \;,_0 \vdash A' = A_1'' :: \Omega$ (Lemma 9.4.10) $\Rightarrow \;,_0 \vdash A' = \mathcal{S}(A, x, \rho) :: \Omega$.

$\rho = \rho'.2$: $\Rightarrow \exists, _0', x', A_1, A_1'.\; ,_0', \Sigma x'{:}A_1.\, A_0 = \;,([\text{self}{=}x]A) \Downarrow \rho'$ and $,'', \Sigma x'{:}A_1'.\, A' = \;,A \Downarrow \rho'$ where $x' \neq x$ and $x' \notin \text{dom}(,'')\cup\text{dom}(,_0')$ (Lemma 11.2.2).
By Corollary 11.2.6 and Lemma 6.5.3,
$\exists A_1'', A_2''.\; \mathcal{S}(A, x, \rho') = \Sigma x{:}A_1''.\, A_2''$ and $\mathcal{S}(A, x, \rho) = [x\rho'.1/x']A_2''$.

Applying part one of the induction hypothesis on ρ' gives
$,_0' \vdash \Sigma x'{:}A_1'.\, A' = \mathcal{S}(A, x, \rho') :: \Omega \Rightarrow \;,_0' \vdash \Sigma x'{:}A_1'.\, A' = \Sigma x{:}A_1''.\, A_2'' :: \Omega$
$\Rightarrow \;,_0', x'{:}A_1' \vdash A' = A_2'' :: \Omega$ (Lemma 9.4.10).

By P-INIT, Lemma 6.3.5, and Corollary 11.2.11, $, \vdash [\text{self}{=}x]A \leq A \Rightarrow \;,_0' \vdash A_1 \leq A_1'$ (Lemma 11.2.12 and Lemma 11.3.3) $\Rightarrow \;,_0', x'{:}A_1 \vdash A' = A_2'' :: \Omega$ (Theorem 10.4.7).
By P-INIT, P-MOVE, E-REFL, Lemma 6.3.5, and Theorem 9.4.4,
$,_0', x'{:}A_1 \vdash [x\rho'.1/x']A_2'' :: \Omega$

By Lemmas 11.3.7, 11.3.6, 11.2.5, 11.3.3, and 11.2.2 combined with the induction hypothesis on paths starting with $\rho'.1$, we have that if
$,_0', x'{:}A_1;,'' \vdash x\rho'.1\rho_2! :: K$ then $,_0', x'{:}A_1;,'' \vdash x\rho'.1\rho_2! = x'\rho_2! :: K$.

Hence by Lemma 11.3.8, $,_0', x'{:}A_1 \vdash A_2'' = [x\rho'.1/x']A_2'' :: \Omega$
$\Rightarrow \;,_0', x'{:}A_1 \vdash A' = [x\rho'/x']A_2'' :: \Omega$ (E-TRAN) $\Rightarrow \;,_0 \vdash A' = \mathcal{S}(A, x, \rho) :: \Omega$.

□

The second key result (Corollary 11.3.12, below) is that if x is bound to a semi-canonical type A_1 in $,$, $, \vdash A_1 \leq A_2$, and $, \vdash [\text{self}{=}x]A_2 :: \Omega$, then $, \vdash [\text{self}{=}x]A_1 \leq [\text{self}{=}x]A_2$. This result is exactly what is needed to prove the replacement by a subtype result for the term level in the T-VAR case (see the VAR I case of Theorem 11.4.20).

The only hard point in the proof is the inductive case where the ρ component via cleaving of A_1 is $<=A::K>$ and of A_2 is $<K>$. In that case, we need to show that $A = x\rho!$ under the appropriate assignment, call it Γ', since $[\text{self}=x\rho]A_1 = <=A::K>$ and $[\text{self}=x\rho]A_2 = <=x\rho!::K>$. It is easy to show that $\Gamma' \vdash x\rho! = \mathcal{S}(A_1, x, \rho) :: K$. The previous key result (Theorem 11.3.9) can then be used to show that $\Gamma' \vdash A = \mathcal{S}(A_1, x, \rho) :: K$.

Lemma 11.3.10 *If* $\Gamma', A' = \Gamma, A \Downarrow \rho$ *and* $A \in C_{\Gamma}$ *then* $A' \in C_{\Gamma'}$.

Proof: Proved by structural induction on ρ using Lemma 9.5.3. □

Theorem 11.3.11 (Self and subtyping) *Suppose:*

1. $\Gamma = \Gamma_1, x{:}A_1; \Gamma_2$
2. $A_1 \in C_{\Gamma_1}$
3. $\Gamma', A'_1 = \Gamma, A_1 \Downarrow \rho$
4. $\Gamma'', A'_2 = \Gamma, A_2 \Downarrow \rho$
5. $\Gamma_0, [\textit{self}=x\rho]A'_1 = \Gamma, [\textit{self}=x]A_1 \Downarrow \rho$
6. $\Gamma' \vdash A'_1 \leq A'_2$
7. $\Gamma_0 \vdash [\textit{self}=x\rho]A'_2 :: \Omega$

Then $\Gamma_0 \vdash [\textit{self}=x\rho]A'_1 \leq [\textit{self}=x\rho]A'_2$.

Proof: Proved by structural induction on A'_1. By P-INIT and Lemma 6.3.5, $\Gamma \vdash x \Rightarrow A_1$ $\Rightarrow \Gamma \vdash [\text{self}=x]A_1 \leq A_1$ (Corollary 11.2.11) $\Rightarrow \Gamma_0 \vdash [\text{self}=x\rho]A'_1 \leq A'_1$ and $\Gamma_0 \vdash A'_1 \leq A'_2$ (Lemma 11.2.12) $\Rightarrow \Gamma_0 \vdash [\text{self}=x\rho]A'_1 \leq A'_2$.

This handles all the cases for $\lceil A'_2 \rceil \notin \{\Sigma, <K>\}$ ($\Rightarrow [\text{self}=x\rho]A'_2 = A'_2$). By Lemma 6.3.5, Theorem 9.5.5, and Lemma 11.3.10, $A'_1 \in C_{\Gamma'} \Rightarrow \lceil A'_1 \rceil_{\Gamma'} = \lceil A'_1 \rceil$ (Lemma 10.3.9) $\Rightarrow$ $\lceil A'_1 \rceil \leq \lceil A'_2 \rceil_{\Gamma'}$ (Lemma 10.3.11). The remaining cases are thus:

case I: $\lceil A'_2 \rceil = <K>$:
$\Rightarrow \lceil A'_2 \rceil_{\Gamma'} = <K>$ (definition of intrinsic shape and E-REFL)
$\Rightarrow \lceil A'_1 \rceil \leq <K>$. There are two cases:

- $\lceil A'_1 \rceil = <K> \Rightarrow A'_1 = <K> = A'_2 \Rightarrow [\text{self}=x\rho]A'_1 = [\text{self}=x\rho]A'_2$
  $\Rightarrow \Gamma_0 \vdash [\text{self}=x\rho]A'_1 \leq [\text{self}=x\rho]A'_2$ (O-REFL and S-ONE).

– $\lceil A_1' \rceil = {<}{=}{::}K{>} \Rightarrow \exists A_4.\ A_2' = {<}K{>}$ and $A_1' = {<}{=}A_4{::}K{>}$ By Theorem 11.3.9, $,_0 \vdash {<}{=}A_4{::}K{>} = \mathcal{S}(A_1, x, \rho) :: \Omega \Rightarrow\ ,_0 \vdash x\rho \Rightarrow {<}{=}A_4{::}K{>}$ (P-INIT, Lemma 6.3.5, Theorem 9.4.4, E-REFL, and P-MOVE)
$\Rightarrow\ ,_0 \vdash x\rho! = A_4 :: K$ (E-ABBREV, C-EXT-T, Theorem 9.4.4, and E-REFL)
$\Rightarrow\ ,_0 \vdash {<}{=}x\rho!{::}K{>} = {<}{=}A_4{::}K{>} :: \Omega$ (E-TRANS)
$\Rightarrow\ ,_0 \vdash [\text{self}{=}x\rho]{<}{=}A_4{::}K{>} = [\text{self}{=}x\rho]{<}K{>} :: \Omega$ (E-SYM)
$\Rightarrow\ ,_0 \vdash [\text{self}{=}x\rho]A_1' = [\text{self}{=}x\rho]A_2' :: \Omega. \Rightarrow$
$,_0 \vdash [\text{self}{=}x\rho]A_1' \leq [\text{self}{=}x\rho]A_2'$ (S-EQ).

case II: $\lceil A_2' \rceil = \Sigma$:
$\Rightarrow \lceil A_2' \rceil_{,\,'} = \Sigma$ (definition of intrinsic shape and E-REFL) $\Rightarrow \lceil A_1' \rceil = \Sigma \Rightarrow$ $\exists x', A_4, A_5, A_6, A_7$ such that $A_1' = \Sigma x'{:}A_4.\, A_5$ and $A_2' = \Sigma x'{:}A_6.\, A_7$ where $x' \notin$ dom$(,')\cup$dom$(,_0)$. By Lemma 10.5.1, $,' \vdash A_4 \leq A_6$ and $,', x'{:}A_4 \vdash A_5 \leq A_7$. By C-DSUM, DECL-T, and Lemma 6.3.5, $,_0 \vdash [\text{self}{=}x\rho.1]A_6 :: \Omega$ and
$,_0, x'{:}[\text{self}{=}x\rho.1]A_6 \vdash [\text{self}{=}x\rho.2]A_7 :: \Omega$.

Hence, by Lemmas 11.2.2, 11.3.3, and the induction hypothesis on A_4,
$,_0 \vdash [\text{self}{=}x\rho.1]A_4 \leq [\text{self}{=}x\rho.1]A_6$
$\Rightarrow\ ,_0, x'{:}[\text{self}{=}x\rho.1]A_4 \vdash [\text{self}{=}x\rho.2]A_7 :: \Omega$ (Theorem 10.4.7). Hence, by Lemmas 11.2.2, 11.3.3, and the induction hypothesis on A_5,
$,_0, x'{:}[\text{self}{=}x\rho.1]A_4 \vdash [\text{self}{=}x\rho.2]A_5 \leq [\text{self}{=}x\rho.2]A_7 \Rightarrow$
$,_0 \vdash \Sigma x'{:}[\text{self}{=}x\rho.1]A_4.\, [\text{self}{=}x\rho.2]A_5 \leq \Sigma x'{:}[\text{self}{=}x\rho.1]A_6.\, [\text{self}{=}x\rho.2]A_7$
(Theorem 10.5.4) $\Rightarrow\ ,_0 \vdash [\text{self}{=}x\rho]A_1' \leq [\text{self}{=}x\rho]A_2'$.

□

Corollary 11.3.12 (Self and subtyping) *Suppose:*

1. $A_1 \in C_{,\,1}$
2. $,_1, x{:}A_1; ,_2 \vdash A_1 \leq A_2$
3. $,_1, x{:}A_1; ,_2 \vdash [\mathit{self}{=}x]A_2 :: \Omega$

Then $,_1, x{:}A_1; ,_2 \vdash [\mathit{self}{=}x]A_1 \leq [\mathit{self}{=}x]A_2$.

Proof: Follows immediately from Theorem 11.3.11 with $\rho = \epsilon$. □

11.4 Runtime States

In this section I introduce the *semantic system*, an extension of the kernel system that allows tracking the richer runtime states that arise during evaluation. The semantic system differs only at the term level; it extends the set of kernel system terms with *locations* (l) and adds *stores* (S) which map locations to *values* (V). Stores are assigned *store types* (Φ) which map locations to types. As a technical device to simplify the semantics, the primitive functions **roll**, **unroll**, **new**, **get**, and **set** are replaced in the semantic system with versions that are either fully uncurried ($\mathbf{roll}_2(\Leftrightarrow, \Leftrightarrow)$) or curried only in their last argument ($\mathbf{unroll}_1(\Leftrightarrow)$, $\mathbf{new}_1(\Leftrightarrow)$, $\mathbf{get}_1(\Leftrightarrow)$, and $\mathbf{set}_2(\Leftrightarrow, \Leftrightarrow)$). The fully curried versions can be regarded as syntactic sugar for the (partially) uncurried ones:

Definition [Sem] 11.4.1 (Syntactic sugar)

$$
\begin{array}{rcl}
\mathbf{roll} & = & \lambda x{:}{<}\Omega{\Rightarrow}\Omega{>}.\ \lambda x'{:}x!\,(\mathbf{rec}\,x!).\ \mathbf{roll}_2(x!, x') \\
\mathbf{unroll} & = & \lambda x{:}{<}\Omega{\Rightarrow}\Omega{>}.\ \mathbf{unroll}_1(x!) \\
\\
\mathbf{new} & = & \lambda x{:}{<}\Omega{>}.\ \mathbf{new}_1(x!) \\
\mathbf{get} & = & \lambda x{:}{<}\Omega{>}.\ \mathbf{get}_1(x!) \\
\mathbf{set} & = & \lambda x{:}{<}\Omega{>}.\ \lambda x'{:}\mathbf{ref}\,x!.\ \mathbf{set}_2(x!, x')
\end{array}
$$

Similarly, the new primitives can be defined in terms of the old ones:

Definition [Sem] 11.4.2

$$
\begin{array}{rcl}
\mathbf{roll}_2(A, M) & = & (\mathbf{roll}\,{<}A{>})\,M \\
\mathbf{unroll}_1(A) & = & \mathbf{unroll}\,{<}A{>} \\
\\
\mathbf{new}_1(A) & = & \mathbf{new}\,{<}A{>} \\
\mathbf{get}_1(A) & = & \mathbf{get}\,{<}A{>} \\
\mathbf{set}_2(A, M) & = & (\mathbf{set}\,{<}A{>})\,M
\end{array}
$$

The syntax for the semantic system is the same as for the kernel system except for the following changes and additions:

Definition [Sem] 11.4.3 (Changed syntax)

$$\begin{array}{llll}
\textit{Terms} & M & ::= & x \mid \lambda x{:}A.\, M \mid M_1\, M_2 \mid (M_1,\ M_2) \mid M.1 \mid M.2 \mid \\
 & & & {<}A{>} \mid M{<:}A \mid l \mid \mathbf{roll}_2(A, M) \mid \\
 & & & \mathbf{unroll}_1(A) \mid \mathbf{new}_1(A) \mid \mathbf{get}_1(A) \mid \mathbf{set}_2(A, M) \\
 & & & \\
\textit{Values} & V & ::= & \lambda x{:}A.\, M \mid (V_1,\ V_2) \mid {<}A{>} \mid l \mid \mathbf{roll}_2(A, V) \mid \\
 & & & \mathbf{unroll}_1(A) \mid \mathbf{new}_1(A) \mid \mathbf{get}_1(A) \mid \mathbf{set}_2(A, l) \\
 & & & \\
\textit{Store types} & \Phi & ::= & \bullet \mid \Phi; l{:}A \\
\textit{Stores} & S & ::= & \bullet \mid S, l{=}V
\end{array}$$

These changes in syntax require the obvious changes in the free term variables and free constructor variables functions:

Definition [Sem] 11.4.4 (Changes)

$$\begin{array}{rcl}
FTV(l) & = & \emptyset \\
FTV(\mathbf{new}_1(A)) & = & FTV(A) \\
FTV(\mathbf{get}_1(A)) & = & FTV(A) \\
FTV(\mathbf{unroll}_1(A)) & = & FTV(A) \\
FTV(\mathbf{set}_2(A, M)) & = & FTV(A) \cup FTV(M) \\
FTV(\mathbf{roll}_2(A, M)) & = & FTV(A) \cup FTV(M) \\
 & & \\
FCV(l) & = & \emptyset \\
FCV(\mathbf{new}_1(A)) & = & FCV(A) \\
FCV(\mathbf{get}_1(A)) & = & FCV(A) \\
FCV(\mathbf{unroll}_1(A)) & = & FCV(A) \\
FCV(\mathbf{set}_2(A, M)) & = & FCV(A) \cup FCV(M) \\
FCV(\mathbf{roll}_2(A, M)) & = & FCV(A) \cup FCV(M)
\end{array}$$

Values are a special subset of terms (all Vs are Ms) that evaluate to themselves. When a program terminates, the result of its evaluation is always a value. With the exception of the body of a lambda, all components of values are themselves values. The set of values includes user functions ($\lambda x{:}A.\, M$), partially applied primitive functions ($\mathbf{unroll}_1(A)$, $\mathbf{new}_1(A)$, $\mathbf{get}_1(A)$, and $\mathbf{set}_2(A, l)$[1]), pairs of values, reified constructors, locations (the values of reference types), and *rolled* values ($\mathbf{roll}_2(A, V)$, the values of recursive types). Term variables, applications, projections, and coercions are never values.

The semantic system has three term-level judgments:

[1]I choose to make the value syntax reflect the fact that $\mathbf{set}_2(-,-)$'s second argument is always a location in a well-typed computation result; the complexity of the system is uneffected by this choice.

Definition [Sem] 11.4.5 (Judgments)

$\vdash \Phi \; valid$	*valid store type*
$, \vdash_{\Phi} M : A$	*well-typed term*
$\vdash S : \Phi$	*well-typed store*

Note that the semantic system's well-typed term judgment takes an additional parameter, a store type (Φ), which is used to determine the types of locations appearing in the term.

Store types are valid if they never redeclare the type of a location and if all types they assign to locations are valid under the empty assignment:

Definition [Sem] 11.4.6 (Store Type Formation Rules)

$$\vdash \bullet \; valid \tag{ST-EMPTY}$$

$$\frac{\vdash \Phi \; valid \qquad \bullet \vdash A :: \Omega \qquad l \notin dom(\Phi)}{\vdash \Phi, l{:}A \; valid} \tag{ST-EXTEND}$$

The rules for the semantic system's well-typed term judgment follow. The LAM, APP, PAIR, FST, SND, SUMP, and COERCE rules are unchanged from the kernel system aside from the addition of the extra store-type parameter. The VAR and REIFY rules have an extra precondition $\vdash \Phi$ valid added to ensure that if $, \vdash_{\Phi} M : A$ then $\vdash \Phi$ valid (see Lemma 11.4.12). The NEW, GET, SET, ROLL, and UNROLL rules are modified to handle the (partially) uncurried versions of the primitive functions. The previous versions of these rules (using the curried versions) are easily verified to be derived rules in the semantic system. The SM-LABEL rule is the only completely new rule; it assigns the type **ref** $\Phi(l)$ to location l.

Definition [Sem] 11.4.7 (Term Formation Rules)

$$\frac{\vdash \Phi \; valid \qquad , \vdash , (x) = A :: \Omega}{, \vdash_{\Phi} x : [self{=}x]A} \tag{SM-VAR}$$

$$\frac{, , x{:}A \vdash_{\Phi} M : A'}{, \vdash_{\Phi} \lambda x{:}A.\, M : \Pi x{:}A.\, A'} \tag{SM-LAM}$$

$$\frac{, \vdash_{\Phi} M_1 : A_2 {\to} A \qquad , \vdash_{\Phi} M_2 : A_2}{, \vdash_{\Phi} M_1 M_2 : A} \tag{SM-APP}$$

$$\frac{,\ \vdash_\Phi M_1 : A_1 \qquad ,\ \vdash_\Phi M_2 : A_2}{,\ \vdash_\Phi (M_1,\, M_2) : (A_1, A_2)} \qquad (\text{SM-PAIR})$$

$$\frac{,\ \vdash_\Phi M : \Sigma x{:}A_1.\, A_2}{,\ \vdash_\Phi M.1 : A_1} \qquad (\text{SM-FST})$$

$$\frac{,\ \vdash_\Phi M : (A_1, A_2)}{,\ \vdash_\Phi M.2 : A_2} \qquad (\text{SM-SND})$$

$$\frac{\vdash \Phi\ \mathit{valid} \qquad ,\ \vdash A :: K}{,\ \vdash_\Phi <A> : <=A::K>} \qquad (\text{SM-REIFY})$$

$$\frac{,\ \vdash_\Phi M : A' \qquad ,\ \vdash A' \leq A}{,\ \vdash_\Phi M : A} \qquad (\text{SM-SUMP})$$

$$\frac{,\ \vdash_\Phi M : A}{,\ \vdash_\Phi M{<:}A : A} \qquad (\text{SM-COERCE})$$

$$\frac{\vdash ,\ \ \mathit{valid} \qquad \vdash \Phi\ \mathit{valid} \qquad l \in \mathit{dom}(\Phi)}{,\ \vdash_\Phi l : \mathbf{ref}\,(\Phi(l))} \qquad (\text{SM-LABEL})$$

$$\frac{\vdash \Phi\ \mathit{valid} \qquad ,\ \vdash A :: \Omega}{,\ \vdash_\Phi \mathbf{new}_1(A) : A{\rightarrow}\mathbf{ref}\,A} \qquad (\text{SM-NEW})$$

$$\frac{\vdash \Phi\ \mathit{valid} \qquad ,\ \vdash A :: \Omega}{,\ \vdash_\Phi \mathbf{get}_1(A) : \mathbf{ref}\,A{\rightarrow}A} \qquad (\text{SM-GET})$$

$$\frac{\vdash \Phi\ \mathit{valid} \qquad ,\ \vdash A :: \Omega{\Rightarrow}\Omega}{,\ \vdash_\Phi \mathbf{unroll}_1(A) : \mathbf{rec}\,A{\rightarrow}A\,(\mathbf{rec}\,A)} \qquad (\text{SM-UNROLL})$$

$$\frac{,\ \vdash_\Phi M : \mathbf{ref}\,A}{,\ \vdash_\Phi \mathbf{set}_2(A, M) : A{\rightarrow}A} \qquad (\text{SM-SET})$$

$$\frac{,\ \vdash_\Phi M : A\,(\mathbf{rec}\,A)}{,\ \vdash_\Phi \mathbf{roll}_2(A, M) : \mathbf{rec}\,A} \qquad (\text{SM-ROLL})$$

Because these rules are so similar to the kernel system ones, the proofs for many propositions are essentially the same for both systems. Accordingly, I present each such proposition only once; these results, marked with [(Sem)], hold both in the semantics system and (once the store types are removed) in the kernel system.

Any well-typed term in the kernel system is also a well-typed term in the semantic system taking Definition 11.4.1 into account:

Lemma 11.4.8 *If* $, \vdash M : A$ *then* $, \vdash_{\bullet} M : A$.

The reverse is also true if we limit ourselves to terms that do not contain locations and use Definition 11.4.2:

Lemma [Sem] 11.4.9 *If* $, \vdash_{\Phi} M : A$ *and* M *contains no locations then* $, \vdash M : A$.

A store S has valid store type Φ iff it assigns a unique value V of type $\Phi(l)$ under the empty assignment and store type Φ to each location l mapped by Φ:

Definition [Sem] 11.4.10 (Store Formation Rules)

$$\frac{\begin{array}{c}\vdash \Phi \ \mathit{valid} \\ |S| = |\Phi| \qquad \forall l \in \mathit{dom}(\Phi).\ \ \bullet \vdash_{\Phi} S(l) : \Phi(l)\end{array}}{\vdash S : \Phi} \qquad \text{(STORE)}$$

As with assignments, stores and store types can be treated as as partial functions:

Definition [Sem] 11.4.11 (Store (types) regarded as partial functions)

$$\begin{array}{rcll} \mathit{dom}(\bullet) & = & \emptyset & \\ \mathit{dom}(\Phi, l{:}A) & = & \mathit{dom}(\Phi) \cup \{l\} & \\ & & & \\ (\Phi_1; l{:}A; \Phi_2)(l) & = & A & (l \notin \mathit{dom}(\Phi_1)) \\ & & & \\ \mathit{dom}(\bullet) & = & \emptyset & \\ \mathit{dom}(S, l{=}V) & = & \mathit{dom}(S) \cup \{l\} & \\ & & & \\ (S_1; l{=}V; S_2)(l) & = & V & (l \notin \mathit{dom}(S_1)) \end{array}$$

Lemma [Sem] 11.4.12 (Store (type) properties)

1. $\mathit{dom}(\Phi; \Phi') = \mathit{dom}(\Phi) \cup \mathit{dom}(\Phi')$
2. $\mathit{dom}(S; S') = \mathit{dom}(S) \cup \mathit{dom}(S')$
3. *If* $l{:}A \in \Phi$ *then* $l \in \mathit{dom}(\Phi)$.
4. *If* $l{=}V \in S$ *then* $l \in \mathit{dom}(S)$.

The usual propositions on judgments hold for the semantic and kernel systems' term-level judgments:

Lemma [Sem] 11.4.13 (Structural properties)

1. *If* $\vdash \Phi$ *valid and* $l \in dom(\Phi)$ *then* $\bullet \vdash \Phi(l) :: \Omega$.
2. *If* $\vdash S : \Phi$ *then* $\vdash \Phi$ *valid.*
3. *If* $, \vdash_{\Phi} M : A$ *then* $\vdash \Phi$ *valid.*

Lemma [Sem] 11.4.14 (Valid store (type) properties)
Suppose $\vdash \Phi$ *valid and* $\vdash S : \Phi'$. *Then:*

1. $dom(S) = dom(\Phi')$
2. *If* $l{:}A \in \Phi$ *then* $\Phi(l) = A$.
3. *If* $l{=}V \in S$ *then* $S(l) = V$.

Proof: Inspection of the typing rules for store (types) reveal that valid store (types) never redeclare/rebind labels. Thus, we have that $|\Phi'| = |\mathrm{dom}(\Phi')|$ and $|S| = |\mathrm{dom}(S)|$ $\Rightarrow \mathrm{dom}(S) = \mathrm{dom}(\Phi')$ by STORE. □

Theorem [(Sem)] 11.4.15 (Validity of Term Types)
If $, \vdash_{\Phi} M : A$ *then* $, \vdash A :: \Omega$.

Proof: Proved by structural induction on the derivation. Interesting cases:

VAR: Given $, \vdash , (x) = A :: \Omega \Rightarrow , \vdash x \Rightarrow A$ (Lemma 6.3.5, P-INIT, and P-MOVE) $\Rightarrow$ $, \vdash [\mathrm{self}{=}x]A :: \Omega$ (Theorem 11.2.10).

SND: Given $, \vdash_{\Phi} M : \Pi x{:}A_1.\, A_2$, $x \notin \mathrm{FTV}(A_2) \Rightarrow , \vdash \Pi x{:}A_1.\, A_2 :: \Omega$ (induction hypothesis) $\Rightarrow , , x{:}A_1 \vdash A_2 :: \Omega$ (C-DSUM) $\Rightarrow , \vdash A_2 :: \Omega$ (Theorem 9.6.1).

SUMP: Follows immediately from Lemma 10.2.2.

SM-NEW: Using P-INIT, C-EXT-O, C-DFUN, C-REF, and C-APP it is easy to prove that $\bullet \vdash \forall\alpha{::}\Omega.\alpha{\to}\mathbf{ref}\,\alpha :: \Omega$ Using weakening (Theorem 9.4.1) then, we get the desired result: $, \vdash \forall\alpha{::}\Omega.\alpha{\to}\mathbf{ref}\,\alpha :: \Omega$.

SM-LABEL: Follows immediately from Lemma 11.4.13, Theorem 9.4.1, Lemma 6.3.5, C-REF, and C-APP.

□

Lemma [(Sem)] 11.4.16 *If* $, \vdash_{\Phi} M : A$ *then* $FTV(M) \subseteq FTV(,)$*.*

Proof: By structural induction on the derivation using Theorem 6.5.6, Lemma 6.3.5, Theorem 11.4.15, and Lemma 10.2.2 as needed. □

Theorem [(Sem)] 11.4.17 (Weakening)
Suppose $\vdash , ; , '$ *valid and* $dom(, ') \cap dom(, '') = \emptyset$*. Then if* $, ; , '' \vdash_{\Phi} M : A$ *then* $, ; , '; , '' \vdash_{\Phi} M : A$*.*

Proof: By structural induction on the derivation of $, ; , '' \vdash_{\Phi} M : A$, using Theorem 9.4.1, Theorem 10.2.3, Lemma 6.3.5, and Lemma 9.4.2 as needed. □

Theorem [Sem] 11.4.18 (Weakening)
Suppose $\vdash \Phi; \Phi'$ *valid. Then if* $, \vdash_{\Phi} M : A$ *then* $, \vdash_{\Phi;\Phi'} M : A$*.*

Proof: By structural induction on the derivation of $, \vdash_{\Phi} M : A$. The only interesting case is SM-LABEL which is handled by the fact that $l \in \mathrm{dom}(\Phi) \Rightarrow l \in \mathrm{dom}(\Phi; \Phi')$ and $\Phi(l) = (\Phi; \Phi')(l)$ (definition of store type as a partial function). □

Theorem [(Sem)] 11.4.19 (Strengthening)
Suppose $x \notin FTV(, ') \cup FTV(M) \cup FTV(A)$*. Then, if* $, , x{:}A'; , ' \vdash_{\Phi} M : A$*, then* $, ; , ' \vdash_{\Phi} M : A$*.*

Proof: By structural induction on the derivation using Theorem 9.6.1, Corollary 9.6.2, Lemma 10.2.4, Lemma 6.3.5, and Lemma 9.4.2 as needed. □

Theorem [(Sem)] 11.4.20 (Replacement by a subtype)
If $, \vdash A_2 \leq A_1$ *and* $, , x{:}A_1; , ' \vdash_{\Phi} M : A$ *then* $, , x{:}A_2; , ' \vdash_{\Phi} M : A$*.*

Proof: By structural induction on the derivation of $, , x{:}A_1; , ' \vdash_{\Phi} M : A$ using Theorem 10.4.7 as needed. The only interesting case is as follows:

VAR I: Here $, , x{:}A_1; , ' \vdash_{\Phi} x : [\mathrm{self}{=}x]A$ derived via rule T/SM-VAR from
$, , x{:}A_1; , ' \vdash (, , x{:}A_1; , ')(x) = A :: \Omega \Rightarrow , , x{:}A_1; , ' \vdash [\mathrm{self}{=}x]A :: \Omega$
(Theorem 11.4.15) $\Rightarrow , , x{:}A_2; , ' \vdash [\mathrm{self}{=}x]A :: \Omega$ and
$, , x{:}A_2; , ' \vdash (, , x{:}A_1; , ')(x) = A :: \Omega$ (Theorem 10.4.7) $\Rightarrow \vdash , , x{:}A_2; , '$ valid and
$, , x{:}A_2; , ' \vdash A_1 = A :: \Omega$ (Lemma 6.3.5 and Theorem 9.4.2)

$\Rightarrow$ $, , x{:}A_2; , ' \vdash A_2 \leq A_1$ (Theorem 10.2.3) $\Rightarrow$ $, , x{:}A_2; , ' \vdash A_2 \leq A$ (S-EQ and E-TRAN).

By Theorem 9.5.10 and Lemma 10.2.2, $\exists A_2'.\ , \vdash A_2 = A_2' :: \Omega$ and $A_2' \in C_{,} \Rightarrow$ $, , x{:}A_2'; , ' \vdash [\text{self}{=}x]A :: \Omega$ (Theorem 9.4.6), $, , x{:}A_2'; , ' \vdash A_2 \leq A$ (Theorem 10.2.5), and $, , x{:}A_2'; , ' \vdash A_2 = A_2' :: \Omega$ (Theorem 10.2.3 and Lemma 6.3.5)
$\Rightarrow$ $, , x{:}A_2'; , ' \vdash A_2' \leq A$ (E-SYM, S-EQ, and S-TRAN). Hence, by Corollary 11.3.12, $, , x{:}A_2'; , ' \vdash [\text{self}{=}x]A_2' \leq [\text{self}{=}x]A$.

By T/SM-VAR and Theorem 9.4.6, $, , x{:}A_2; , ' \vdash_\Phi x : [\text{self}{=}x]A_2'$
$\Rightarrow$ $, , x{:}A_2; , ' \vdash_\Phi x : [\text{self}{=}x]A$ (T/SM-SUMP).

□

11.5 Properties of Values

In this section I establish a number of properties of values that I shall need to prove soundness. In order to do type inference in the presence of subsumption, it is necessary to be able to compute *minimal* types of terms:

Definition [Sem] 11.5.1 *A is a minimal type for M under , and Φ ($A \in Min_{, ,\Phi}(M)$) iff:*

1. $, \vdash_\Phi M : A$
2. *If* $, \vdash_\Phi M : A'$ *then* $, \vdash A \leq A'$.

I shall need the minimal types of the primitive function values in order to determine the types of their arguments in applications:

Lemma [Sem] 11.5.2 (Minimal types of selected values)
Suppose $, \vdash_\Phi V : A$. *Then:*

1. *If* $V = \texttt{<}A'\texttt{>}$ *then* $\exists K.\ \texttt{<=}A'{::}K\texttt{>} \in Min_{, ,\Phi}(V)$.
2. *If* $V = l$ *then* $\mathbf{ref}\,\Phi(l) \in Min_{, ,\Phi}(V)$.
3. *If* $V = \mathbf{new}_1(A)$ *then* $A \rightarrow \mathbf{ref}\,A \in Min_{, ,\Phi}(V)$.
4. *If* $V = \mathbf{get}_1(A)$ *then* $\mathbf{ref}\,A \rightarrow A \in Min_{, ,\Phi}(V)$.
5. *If* $V = \mathbf{set}_2(A, l)$ *then* $A \rightarrow A \in Min_{, ,\Phi}(V)$.

6. *If* $V = \mathbf{roll}_2(A, V)$ *then* $\mathbf{rec}\, A \in \mathit{Min}_{,\,,\Phi}(V)$.

7. *If* $V = \mathbf{unroll}_1(A)$ *then* $\mathbf{rec}\, A {\rightarrow} A\, (\mathbf{rec}\, A) \in \mathit{Min}_{,\,,\Phi}(V)$.

Proof: Any derivation of $, \vdash_\Phi V : A$ for the above values must consist of an application of SM-REIFY/SM-LABEL/SM-NEW/SM-GET/SM-SET/SM-ROLL/SM-UNROLL respectively followed by zero or more applications of SM-SUMP. Hence, A must be a supertype under , for the respective listed type by S-TRAN. □

This information can also be used to compute the intrinsic shapes of values' types:

Lemma [(Sem)] 11.5.3 *If* $, \vdash_\Phi M : A$ *and* $\lceil A \rceil \neq ?$ *then* $\lceil A \rceil_{,} = \lceil A \rceil$.

Proof: By Theorem 11.4.15 and E-REFL, $, \vdash A = A :: \Omega \Rightarrow \lceil A \rceil_{,} = \lceil A \rceil$ (definition of intrinsic shape). □

Lemma [Sem] 11.5.4 (Intrinsic shapes of values' types)
Suppose $, \vdash_\Phi V : A$. *Then:*

1. *If* $V = {<}A'{>}$ *then* $\exists K.\ {<}{=}{::}K{>} \leq \lceil A \rceil_{,}$.

2. *If* $V = l$ *then* $\lceil A \rceil_{,} = \mathbf{refapp}$.

3. *If* $V = \mathbf{roll}_2(A, V)$ *then* $\lceil A \rceil_{,} = \mathbf{recapp}$.

4. *If* $V = \mathbf{new}_1(A)$, $\mathbf{get}_1(A)$, $\mathbf{set}_2(A, l)$, *or* $\mathbf{unroll}_1(A)$ *then* $\lceil A \rceil_{,} = \Pi$.

5. *If* $V = \lambda x{:}A'.\, V'$ *then* $\lceil A \rceil_{,} = \Pi$.

6. *If* $V = (V_1, V_2)$ *then* $\lceil A \rceil_{,} = \Sigma$.

Proof: For parts 1-4, Lemma 11.5.2, provides a minimal type A' for V under , such that $\lceil A' \rceil \neq ? \Rightarrow \lceil A' \rceil_{,} = \lceil A' \rceil$ (Lemma 11.5.3 and definition of minimal type) and $, \vdash A' \leq A$ (definition of minimal type) $\Rightarrow \lceil A' \rceil \leq \lceil A \rceil_{,}$ (Corollary 10.3.12) $\Rightarrow \exists K.\ {<}{=}{::}K{>} \leq \lceil A \rceil_{,}$ for part 1 and $\lceil A' \rceil = \lceil A \rceil_{,}$ for parts 2-4.

For parts 5/6, any derivation of $, \vdash_\Phi V : A$ for the values in question must consist of an application of SM-LAM/SM-PAIR respectively followed by zero or more applications of SM-SUMP. Hence, A must be a supertype under , of a dependent function/pair type by S-TRAN $\Rightarrow$ A's intrinsic shape under , must be Π/Σ (Corollary 10.3.12 and Lemma 11.5.3). □

Before I describe the last property of values I shall need, I first need to introduce some definitions. A *transparent* constructor is one that contains no direct components (i.e., the result of a cleaving operation) of the form ${<}K{>}$ for some K:

Definition 11.5.5 (Transparent constructors)
A constructor A is a transparent *constructor iff either:*

1. $\forall K.\ \lceil A \rceil \notin \{\Sigma, {<}K{>}\}$

2. $\exists A_1, A_2.\ A = \Sigma x{:}A_1.\ A_2$ *and both A_1 and A_2 are themselves transparent constructors.*

Because the self function only adds information to direct components of the form $<K>$, the self function leaves transparent constructors unchanged:

Lemma 11.5.6 *If A is a transparent constructor then $[self{=}x\rho]A = A$.*

A *disconnected* constructor is one in which no direct component refers back to a previous component:

Definition 11.5.7 (Disconnected constructors)
A constructor A is a disconnected *constructor iff either:*

1. $\lceil A \rceil \neq \Sigma$

2. $\exists A_1, A_2.\ A = \Sigma x{:}A_1.\ A_2$*, $x \notin FTV(A_2)$, and both A_1 and A_2 are themselves disconnected constructors.*

The property of being a transparent and disconnected constructor is preserved by selection:

Lemma 11.5.8 *If A is a transparent and disconnected constructor and $\mathcal{S}(A, x\rho, \rho')$ exists then $\mathcal{S}(A, x\rho, \rho')$ is a transparent and disconnected constructor.*

Proof: Proved by structural induction on ρ'. Example case:

Snd: Here $\rho' = .2\rho''$ and $A = (A_1, A_2) \Rightarrow \mathcal{S}(A, x\rho, .2) = A_2 \Rightarrow \mathcal{S}(A, x\rho, .2\rho'') = \mathcal{S}(\mathcal{S}(A, x\rho, .2), x\rho.2, \rho'') = \mathcal{S}(A_2, x\rho.2, \rho'')$ (Lemma 6.5.3). By the definitions of transparent and disconnected constructors, A_2 is a transparent and disconnected constructor. Hence, by the inductive hypothesis, $\mathcal{S}(A_2, x\rho.2, \rho'')$ is a transparent and disconnected constructor $\Rightarrow \mathcal{S}(A, x\rho, \rho')$ is a transparent and disconnected constructor.

$\Box$

The importance of the property of being a transparent disconnected constructor is that when a term variable has a transparent disconnected semi-canonical type, all paths involving it are defined transparently rather than opaquely:

Lemma [Sem] 11.5.9 *If $A \in C_\Gamma$ and A is a transparent disconnected constructor then $\Gamma, x{:}A; \Gamma' \vdash x\rho \Rightarrow <K>$ is impossible.*

Proof: By Lemma 9.5.9 and E-SYM, $\Gamma, x{:}A; \Gamma' \vdash \mathcal{S}(A, x, \rho) = <K> :: \Omega$ and $\mathcal{S}(A, x, \rho) \in C_{\Gamma, x:A; \Gamma'} \Rightarrow \mathcal{S}(A, x, \rho) = <K>$ (Lemma 9.5.7) $\Rightarrow \mathcal{S}(A, x, \rho)$ is not a transparent constructor. But this this contradicts Lemma 11.5.8, so this is impossible. □

All values can be assigned such a type as a minimal type:

Theorem [Sem] 11.5.10 *If $\Gamma \vdash_\Phi V : A$ then there exists a type A' such that:*

1. $\Gamma \vdash A' \leq A$
2. $\Gamma \vdash_\Phi V : A'$
3. $A' \in C_\Gamma$
4. *A' is a transparent constructor*
5. *A' is a disconnected constructor*

Proof: Proved by structural induction on V. Cases:

REIFY: Here $V = <A_1> \Rightarrow \exists K.\ <=A_1::K> \in \mathrm{Min}_{\Gamma,\Phi}(V)$. (Lemma 11.5.2) $\Rightarrow \Gamma \vdash_\Phi V : <=A_1::K>$ and $\Gamma \vdash <=A_1::K> \leq A$ (definition of minimal type). By Theorem 11.4.15 and Theorem 9.5.10, $\exists A'.\ \Gamma \vdash <=A_1::K> = A' :: \Omega$ and $A' \in C_\Gamma$ $\Rightarrow \Gamma \vdash_\Phi V : A'$ and $\Gamma \vdash A' \leq A$ (S-EQ, S-TRAN, and SM-SUMP) and $\lceil A' \rceil = \lceil A' \rceil_\Gamma = \lceil <=A_1::K> \rceil = <=::K>$ (Lemma 10.3.9 and definition of intrinsic shape) $\Rightarrow A'$ is a transparent and disconnected constructor.

PAIR: Here $V = (V_1, V_2)$. WLOG, $\Gamma \vdash_\Phi V : A$ was derived without using the SM-SUMP rule $\Rightarrow \exists A_1, A_2.\ A = (A_1, A_2)$, $\Gamma \vdash_\Phi V_1 : A_1$, and $\Gamma \vdash_\Phi V_2 : A_2$ (SM-PAIR). Hence, by the induction hypothesis, $\forall i \in \{1, 2\}.\ \exists A_i'.\ \Gamma \vdash A_i' \leq A_i$, $\Gamma \vdash_\Phi V_i : A_i'$, $A_i' \in C_\Gamma$, and A_i' is a transparent and disconnected constructor. Let $A' = \Sigma x{:}A_1'.\, A_2'$, $x \notin \mathrm{FTV}(A_2') \cup \mathrm{dom}(\Gamma) \Rightarrow A'$ is a transparent and disconnected constructor, $\Gamma \vdash_\Phi (V_1, V_2) : A'$ (SM-PAIR and DECL-T), $\Gamma, x{:}A_1' \vdash A_2' \leq A_2$ and $\Gamma, x{:}A_1 \vdash_\Phi A_2 : \Omega$ (Theorem 9.4.1 and DECL-T), and $A_2' \in C_{\Gamma, x:A_1'}$ (Lemma 9.5.5 and DECL-T) $\Rightarrow \Gamma \vdash A' \leq A$ (Theorem 10.5.4) and $A' \in C_\Gamma$ (Lemma 9.5.4).

OTHER: Here V does not have the form $<A_1>$ or the form (V_1, V_2). By Theorem 11.4.15 and Theorem 9.5.10, $\exists A'.\ \Gamma \vdash A = A' :: \Omega$ and $A' \in C_\Gamma \Rightarrow \Gamma \vdash A' \leq A$ and $\Gamma \vdash A \leq A'$ (E-SYM and S-EQ) and $\lceil A' \rceil_\Gamma = \lceil A' \rceil$ (Lemma 10.3.9) $\Rightarrow \Gamma \vdash_\Phi V : A'$ (SM-SUMP) $\Rightarrow \forall K.\ \lceil A' \rceil \notin \{\Sigma, <K>\} \Rightarrow A'$ is a transparent and disconnected constructor.

□

11.6 Value Substitution

In this section I introduce *value substitution* ($[V/x]\Leftrightarrow$) which involves substituting values for term variables in constructors, assignments, and terms. Value substitution is the analog of constructor substitution in reducing applications at the term level ($(\lambda x{:}A.\,M)\,V$ reduces to $[V/x]M$). Before I can define value substitution, I shall need some auxiliary definitions.

The definition of selection can be extended to values:

Definition [Sem] 11.6.1 (Value selection)

$$\begin{array}{rcl} V \downarrow \epsilon & = & V \\ \\ (V_1,\, V_2) \downarrow .1\rho' & = & V_1 \downarrow \rho' \\ (V_1,\, V_2) \downarrow .2\rho' & = & V_2 \downarrow \rho' \end{array}$$

Note that the definition is simpler for values because of the lack of internal names for components. Like selection on constructors, selection on values can also be done in steps:

Lemma 11.6.2 *if $V \downarrow .i\rho'$ or $(V \downarrow .i) \downarrow \rho'$ exists then*

$$V \downarrow .i\rho' = (V \downarrow .i) \downarrow \rho'$$

Proof: By inspection of the definition of value selection. □

Lemma 11.6.3 *If $V \downarrow \rho_1\rho_2$ exists then*

$$V \downarrow \rho_1\rho_2 = (V \downarrow \rho_1) \downarrow \rho_2$$

Proof: By structural induction on ρ_1. The non-basis case is as follows:

$$\begin{array}{lll} & V \downarrow .i\rho\rho_2 & \\ = & (V \downarrow .i) \downarrow \rho\rho_2 & \text{(Lemma 6.5.2)} \\ = & ((V \downarrow .i) \downarrow \rho) \downarrow \rho_2 & \text{(induction hypothesis)} \\ = & (V \downarrow .i\rho) \downarrow \rho_2 & \text{(Lemma 6.5.2)} \end{array}$$

□

The type of a value selection by ρ on V can be related to V's type if it is a transparent disconnected type that can be selected on by ρ:

Lemma [Sem] 11.6.4 *If* $, \vdash_\Phi V : A$, *A a transparent disconnected constructor, and* $\mathcal{S}(A, x, \rho)$ *exists then* $, \vdash_\Phi (V \downarrow \rho) : \mathcal{S}(A, x, \rho)$.

Proof: Proved by structural induction on ρ. Example case:

Snd: Here $\rho = .2\rho'$ and $A = (A_1, A_2) \Rightarrow V$ has form (V_1, V_2) (Lemma 11.5.4, definition of intrinsic shape, Theorem 9.4.3, and E-REFL) $\Rightarrow V \downarrow .2 = V_2$ and $\mathcal{S}(A, x, .2) = A_2$ $\Rightarrow V \downarrow .2\rho' = (V \downarrow .2) \downarrow \rho' = V_2 \downarrow \rho'$ (Lemma 11.6.3) and $\mathcal{S}(A, x, .2\rho') = \mathcal{S}(\mathcal{S}(A, x, .2), x, \rho') = \mathcal{S}(A_2, x, \rho')$ (Lemma 6.5.3). By the definition of a transparent disconnected constructor, A_2 is a transparent disconnected constructor. Inspection of the typing rules shows that $, \vdash_\Phi V : A$ must be derived using SM-PAIR followed by zero or more occurrences of SM-SUMP $\Rightarrow \exists A_1', A_2'.\ , \vdash (A_1', A_2') \leq (A_1, A_2)$, $, \vdash_\Phi V_1 : A_1'$, and $, \vdash_\Phi V_2 : A_2' \Rightarrow\ , , x{:}A_1' \vdash A_2' \leq A_2$ for some x such that $x \notin \mathrm{FTV}(A_2) \cup \mathrm{FTV}(A_2') \cup \mathrm{dom}(,)$ (Lemma 10.5.1) $\Rightarrow\ , \vdash A_2' \leq A_2$ (Theorem 10.2.4) $\Rightarrow\ , \vdash_\Phi V_2 : A_2$ (SM-SUMP). Hence, by the induction hypothesis, $, \vdash_\Phi (V_2 \downarrow \rho') : \mathcal{S}(A_2, x, \rho') \Rightarrow\ , \vdash_\Phi (V \downarrow \rho) : \mathcal{S}(A, x, \rho)$.

□

The *extraction* operator on values is a partial function that converts from reified constructors to their component constructor:

Definition [Sem] 11.6.5 (Extraction)

$$<A>! = A$$

Again, the result of the extraction operator on a value can be related to that value's type:

Lemma [Sem] 11.6.6 *If* $, \vdash_\Phi V : <=A::K>$ *then* $, \vdash V! = A :: K$.

Proof: By Lemma 11.5.3, $\lceil <=A::K> \rceil_{,} = <=::K> \Rightarrow V$ has form $<A'>$ for some A' (Lemma 11.5.4) $\Rightarrow V! = A'$ and $\exists K'.\ , \vdash <=A'::K'> \leq <=A::K>$ (Lemma 11.5.2 and definition of minimal type) $\Rightarrow K = K'$ and $, \vdash <=A'::K> = <=A::K> :: \Omega$ (Corollary 10.3.14, Lemma 10.2.2, E-REFL, and definition of intrinsic shape) $\Rightarrow , \vdash A' = A :: K$ (Lemma 9.4.10) $\Rightarrow\ , \vdash V! = A :: K$. □

These results can be combined to determine the type of a value selection followed by an extraction:

Theorem [Sem] 11.6.7
If $, \vdash_\Phi V : A$, *A a transparent disconnected constructor,* $A \in C_{,}$ *, and* $, , x{:}A; ,' \vdash x\rho \Rightarrow <=A'::K>$ *then* $, , x{:}A; ,' \vdash (V \downarrow \rho)! = A' :: K$.

Proof: By Lemma 9.5.9 and E-SYM, $, , x{:}A; , ' \vdash \mathcal{S}(A, x, \rho) = \texttt{<=}A'{::}K\texttt{>} :: \Omega$. By Lemma 11.6.4 and Theorem 11.4.17, $, , x{:}A; , ' \vdash_{\Phi} (V \downarrow \rho) : \mathcal{S}(A, x, \rho)$
$\Rightarrow , , x{:}A; , ' \vdash_{\Phi} (V \downarrow \rho) : \texttt{<=}A'{::}K\texttt{>}$ (S-EQ and SM-SUMP)
$\Rightarrow , , x{:}A; , ' \vdash (V \downarrow \rho)! = A' :: K$ (Lemma 11.6.6). □

Using these definitions, I can define value substitution ($[V/x]\Leftrightarrow$) on constructors and assignments; the key idea is to replace $x\rho$ with $(V \downarrow \rho)!$:

Definition [Sem] 11.6.8 (Value substitution on constructors)

$$\begin{array}{rcll}
[V/x]x\rho! & = & (V \downarrow \rho)! & \\
[V/x]x'\rho! & = & x'\rho! & (x' \neq x) \\
\\
[V/x]\lambda\alpha{::}K.\,A & = & \lambda\alpha{::}K.\,[V/x]A & (\alpha' \notin FCV(V)) \\
[V/x]\Pi x'{:}A_1.\,A_2 & = & \Pi x'{:}[V/x]A_1.\,[V/x]A_2 & (x' \neq x,\ x' \notin FTV(V)) \\
[V/x]\Sigma x'{:}A_1.\,A_2 & = & \Sigma x'{:}[V/x]A_1.\,[V/x]A_2 & (x' \neq x,\ x' \notin FTV(V)) \\
\\
[V/x](A_1\,A_2) & = & [V/x]A_1\,[V/x]A_2 & \\
[V/x]\texttt{<=}A'{::}K\texttt{>} & = & \texttt{<=}[V/x]A'{::}K\texttt{>} & \\
\\
[V/x]\alpha & = & \alpha & \\
[V/x]\texttt{<}K\texttt{>} & = & \texttt{<}K\texttt{>} & \\
[V/x]\mathbf{rec} & = & \mathbf{rec} & \\
[V/x]\mathbf{ref} & = & \mathbf{ref} & \\
\\
[V/x](\alpha{::}K) & = & \alpha{::}K & \\
[V/x](x'{:}A') & = & x'{:}[V/x]A' & \\
\\
[V/x]\bullet & = & \bullet & \\
[V/x](, , D) & = & ([V/x],), [V/x]D &
\end{array}$$

Note that because $(V \downarrow \rho)!$ is not always defined, value substitution is a partial function.

Lemma [Sem] 11.6.9 (Free variables)

1. *If $V \downarrow \rho$ exists, then $FTV(V \downarrow \rho) \subseteq FTV(V)$.*
2. *If $V!$ exists, then $FTV(V!) = FTV(V)$.*
3. *If $[V/x]A$ exists and $x \notin FTV(V)$ then $x \notin FTV([V/x]A)$.*

4. If $x \notin FTV(A)$ then $[V/x]A = A$.

If x is assigned a transparent disconnected semi-canonical type, then substitution of a value with that type for x in a constructor results in an equal constructor:

Lemma [Sem] 11.6.10 *If $, \vdash_{\Phi} V : A$, A a transparent disconnected constructor, $A \in C_{,}$, and $, , x{:}A; , ' \vdash A' :: K$ then $, , x{:}A; , ' \vdash A' = [V/x]A' :: K$.*

Proof: Proved by structural induction on the derivation of $, , x{:}A; , ' \vdash A' :: K$. Example cases:

C-DSUM: Here $, , x{:}A; , ' \vdash \Sigma x'{:}A_1.\, A_2 :: \Omega$, $x' \neq x$ and $x' \notin$ FTV(V), derived via rule C-DSUM from $, , x{:}A; , ', x'{:}A_1 \vdash A_2 :: \Omega \Rightarrow , , x{:}A; , ' \vdash A_1 :: \Omega$ (DECL-T).

Hence, by the induction hypothesis, $, , x{:}A; , ' \vdash A_1 = [V/x]A_1 :: \Omega$ and
$, , x{:}A; , ', x'{:}A_1 \vdash A_2 = [V/x]A_2 :: \Omega$
$\Rightarrow , , x{:}A; , ' \vdash \Sigma x'{:}A_1.\, A_2 = \Sigma x'{:}[V/x]A_1.\, [V/x]A_2 :: \Omega$ (E-DSUM)
$\Rightarrow , , x{:}A; , ' \vdash \Sigma x'{:}A_1.\, A_2 = [V/x](\Sigma x'{:}A_1.\, A_2) :: \Omega$

C-EXT-O I: Here $, , x{:}A; , ' \vdash x\rho! :: K$ derived via rule C-EXT-O from
$, , x{:}A; , ' \vdash x\rho \Rightarrow \,<K>$. But, by Lemma 11.5.9, this is impossible so this case cannot exist.

C-EXT-T I: Here $, , x{:}A; , ' \vdash x\rho! :: K$ derived via rule C-EXT-O from
$, , x{:}A; , ' \vdash x\rho \Rightarrow \,<{=}A'{::}K> \,\Rightarrow , , x{:}A; , ' \vdash x\rho! = A' :: K$ (E-ABBREV, E-REFL, Theorem 9.4.4) and $, , x{:}A; , ' \vdash (V \downarrow \rho)! = A' :: K$. (Theorem 11.6.7)
$\Rightarrow , , x{:}A; , ' \vdash x\rho! = (V \downarrow \rho)! :: K$ (E-SYM and E-TRAN)
$\Rightarrow , , x{:}A; , ' \vdash x\rho! = [V/x]x\rho! :: K$

□

Extending value substitution to terms with $[V/x]x$ defined to be V is straightforward:

Definition [Sem] 11.6.11 (Value substitution on terms)

$$\begin{array}{rcll}
[V/x]x & = & V & \\
[V/x]x' & = & x' & (x \neq x') \\
 & & & \\
[V/x]\lambda x{:}A.\, M & = & \lambda x{:}[V/x]A.\, [V/x]M & (x' \neq x,\ x' \notin FTV(V)) \\
 & & & \\
[V/x](M_1\, M_2) & = & [V/x]M_1\, [V/x]M_2 & \\
[V/x](M_1,\, M_2) & = & ([V/x]M_1,\, [V/x]M_2) & \\
[V/x](M.1) & = & ([V/x]M).1 & \\
[V/x](M.2) & = & ([V/x]M).2 & \\
[V/x]{<}A{>} & = & {<}[V/x]A{>} & \\
[V/x](M{<:}A) & = & ([V/x]M){<:}[V/x]A & \\
[V/x]l & = & l & \\
 & & & \\
[V/x]\mathbf{roll}_2(A, M) & = & \mathbf{roll}_2([V/x]A, [V/x]M) & \\
[V/x]\mathbf{unroll}_1(A) & = & \mathbf{unroll}_1([V/x]A) & \\
[V/x]\mathbf{new}_1(A) & = & \mathbf{new}_1([V/x]A) & \\
[V/x]\mathbf{get}_1(A) & = & \mathbf{get}_1([V/x]A) & \\
[V/x]\mathbf{set}_2(A, M) & = & \mathbf{set}_2([V/x]A, [V/x]M) &
\end{array}$$

The previous result on the existence and validity of value substitution on constructors (Lemma 11.6.10) can be extended to value substitution on terms:

Theorem [Sem] 11.6.12 *Suppose* $, \vdash_\Phi V : A$, *A a transparent disconnected constructor, $A \in C_{,}$, and* $, , x{:}A; ,' \vdash_\Phi M : A'$ *then* $, , x{:}A; ,' \vdash_\Phi [V/x]M : [V/x]A'$.

Proof: Proved by structural induction on the derivation of $, , x{:}A; ,' \vdash_\Phi M : A'$. Note that because of Theorem 11.4.15, Lemma 11.6.10, S-EQ, and SM-SUMP, a proof of $, , x{:}A; ,' \vdash_\Phi [V/x]M : A'$ is also sufficient. We prove whichever is most convenient from case to case. Interesting cases:

SM-VAR I: Here $, , x{:}A; ,' \vdash_\Phi x : [\text{self}{=}x]A'$ derived via rule SM-VAR from $\vdash \Phi$ valid and $, , x{:}A; ,' \vdash (, , x{:}A; ,')(x) = A' :: \Omega$. By Lemma 6.3.5, Lemma 9.4.2, and P-INIT, $(, , x{:}A; ,')(x) = A$ and $, , x{:}A; ,' \vdash x \Rightarrow A \Rightarrow , , x{:}A; ,' \vdash A \leq A'$ (S-EQ).
By P-INIT, Lemma 6.3.5, Theorem 9.4.4, E-REFL, P-MOVE, and Theorem 11.2.10, $, , x{:}A; ,' \vdash [\text{self}{=}x]A' :: \Omega$. By Lemma 6.3.5, Theorem 11.4.15, Theorem 11.4.17, and Lemma 9.5.5, $, , x{:}A; ,' \vdash_\Phi V : A$ and $A \in C_{, , x:A; ,'}$. By Theorem 11.3.11 then, $, , x{:}A; ,' \vdash [\text{self}{=}x]A \leq [\text{self}{=}x]A'$
$\Rightarrow , , x{:}A; ,' \vdash A \leq [\text{self}{=}x]A'$ (Lemma 11.5.6) $\Rightarrow , , x{:}A; ,' \vdash_\Phi V : [\text{self}{=}x]A'$ (SM-SUMP) $\Rightarrow , , x{:}A; ,' \vdash_\Phi [V/x]x : [\text{self}{=}x]A'$

SM-VAR II: Here $, ,x{:}A; ,' \vdash_\Phi x' : [\text{self}{=}x']A'$, $x \neq x'$, derived via rule SM-VAR
$\Rightarrow\ , ,x{:}A; ,' \vdash_\Phi [V/x]x' : [\text{self}{=}x']A'$

SM-LAM: Here $, ,x{:}A; ,' \vdash_\Phi \lambda x'{:}A'.\, M : \Pi x'{:}A'.\, A''$, $x' \neq x$ and $x' \notin \text{FTV}(V)$, derived via rule SM-LAM from $, ,x{:}A; ,', x'{:}A' \vdash_\Phi M : A''$
$\Rightarrow\ , ,x{:}A; ,', x'{:}A' \vdash_\Phi [V/x]M : [V/x]A''$ (induction hypothesis)
$\Rightarrow\ , ,x{:}A; ,', x'{:}[V/x]A' \vdash_\Phi [V/x]M : [V/x]A''$ (S-EQ, Theorem 11.4.20, and Lemma 11.6.10)
$\Rightarrow\ , ,x{:}A; ,' \vdash_\Phi \lambda x'{:}[V/x]A'.\, [V/x]M : \Pi x'{:}[V/x]A'.\, [V/x]A''$ (SM-LAM)
$\Rightarrow\ , ,x{:}A; ,' \vdash_\Phi [V/x]\lambda x'{:}A'.\, M : [V/x]\Pi x'{:}A'.\, A''$

SM-REIFY: Here $, ,x{:}A; ,' \vdash_\Phi \langle A'\rangle : \langle{=}A'{::}K\rangle$ derived via rule SM-REIFY from $\vdash \Phi$ valid and $, ,x{:}A; ,' \vdash A' :: K \Rightarrow\ , ,x{:}A; ,' \vdash [V/x]A' :: K$ (Lemma 11.6.10 and Theorem 9.4.3)
$\Rightarrow\ , ,x{:}A; ,' \vdash_\Phi \langle [V/x]A'\rangle : \langle{=}[V/x]A'{::}K\rangle$ (SM-REIFY)
$\Rightarrow\ , ,x{:}A; ,' \vdash_\Phi [V/x]\langle A'\rangle : [V/x]\langle{=}A'{::}K\rangle$

SM-SUMP: Here $, ,x{:}A; ,' \vdash_\Phi M : A'$ derived via rule SM-SUMP from
$, ,x{:}A; ,' \vdash_\Phi M : A''$ and $, ,x{:}A; ,' \vdash A'' \leq A'$
$\Rightarrow\ , ,x{:}A; ,' \vdash [V/x]A'' \leq [V/x]A'$ (Lemma 11.6.10), E-SYM, S-EQ, S-TRAN) and $, ,x{:}A; ,' \vdash_\Phi [V/x]M : [V/x]A''$ (induction hypothesis)
$\Rightarrow\ , ,x{:}A; ,' \vdash_\Phi [V/x]M : [V/x]A'$ (SM-SUMP)

□

Because all values have minimal types that are transparent disconnected semi-canonical types, the previous theorem can be strengthened by dropping the requirements on x's type:

Corollary [Sem] 11.6.13 (Validity of value substitution)
If $, \vdash_\Phi V : A$ *and* $, ,x{:}A; ,' \vdash_\Phi M : A'$ *then* $, ; [V/x], ' \vdash_\Phi [V/x]M : [V/x]A'$.

Proof: By Theorem 11.5.10, $\exists$ a type A'' such that $, \vdash A'' \leq A$, $, \vdash_\Phi V : A''$, $A'' \in C_,$, and A'' is a transparent disconnected constructor. By Theorem 11.4.20,
$, ,x{:}A''; ,' \vdash_\Phi M : A' \Rightarrow\ , ,x{:}A''; ,' \vdash_\Phi [V/x]M : [V/x]A'$
$\Rightarrow\ , ,x{:}A''; [V/x], ' \vdash_\Phi [V/x]M : [V/x]A'$ (repeated use of Lemma 6.3.5, Lemma 11.6.10, and Theorem 9.4.6) $\Rightarrow\ , ; [V/x], ' \vdash_\Phi [V/x]M : [V/x]A'$ (Lemma 11.6.9 and Theorem 11.4.19). □

11.7 Evaluation

In this section I define how evaluation works then prove soundness for the semantic system. Except where stated otherwise, all constructors, terms, and so on in this section are drawn from the semantic system.

Analogous to the constructor context notion I introduced in Section 7.7, a notion of a term context can be introduced:

$$\begin{array}{lll} \textit{Term Contexts}\ \ \mathcal{C} & ::= & [] \mid \lambda x{:}A.\mathcal{C} \mid \mathcal{C}\,M \mid M\,\mathcal{C} \mid (\mathcal{C},\,M) \mid (M,\mathcal{C}) \mid \\ & & \mathcal{C}.1 \mid \mathcal{C}.2 \mid \mathcal{C}{<:}A \mid \mathbf{roll}_2(A,\mathcal{C}) \mid \mathbf{set}_2(A,\mathcal{C}) \end{array}$$

As before, filling a hole in a term context $\mathcal{C}$ with a term M (written $\mathcal{C}[M]$) can incur the capture of free variables in M that are bound at the occurrence of the hole in $\mathcal{C}$. For example, $(\lambda x{:}A.\,[])[x] = \lambda x{:}A.\,x$. Again as before, I write ETV($\mathcal{C}$) for the exposed term variables of a term context. (Since term contexts never bind constructor variables, there is no need for a notion of exposed constructor variables of a term context.)

Type checking of terms is compositional in the sense that if a term is well typed, then so are all its constituent terms:

Lemma [(Sem)] 11.7.1 (Decomposition)
Suppose that $,\ \vdash_\Phi \mathcal{C}[M] : A$ *such that* $ETV(\mathcal{C}) \cap dom(,\ ) = \emptyset$.[2] *Then there exists* $,\,'$ *and* A' *such that:*

- $dom(,\,') = ETV(\mathcal{C})$
- $,\ ;\ ,\,' \vdash_\Phi M : A'$ *by a rule other than T/SM-SUMP*
- *If* $,\ ;\ ,\,' \vdash_{\Phi;\Phi'} M' : A'$ *then* $,\ \vdash_{\Phi;\Phi'} \mathcal{C}[M'] : A$.

Proof: Proved by structural induction on $\mathcal{C}$ using Lemma 11.4.13 and Theorem 11.4.18 as needed. □

Defining an evaluation strategy requires giving both a set of reductions and a way to determine which potential reduction (if any) to do next. I shall specify which reduction to do next by defining syntactically a set of redices (R) representing potential reduction sites and a subset of term contexts, called evaluation contexts (E). I shall prove that any closed well-typed term M is either a value or can be expressed as $E[R]$ in exactly one way.

Evaluation of closed well-typed terms using a compatible valid store (i.e., both the term and the store were typed using the same store type) is then defined as follows: If

[2]The condition on the exposed variables can always be satisfied by alpha-renaming $\mathcal{C}[M]$ appropriately.

M is a value, then it evaluates to itself using any store. Otherwise, if $M = E[R]$, then M evaluates using S to the result of evaluating $E[M']$ using store S' where R reduces using S to M' while yielding the new store S'.

I shall also prove that reductions of well-typed terms produce well-typed terms. When combined with the result about expressing well-typed terms in terms of values, redices, and evaluation contexts, this result will show that my evaluation strategy deterministically chooses which, if any, potential reduction to do next and that it never gets "stuck" in a state where it is trying to evaluate a non-value that has no potential reduction sites.

Definition [Sem] 11.7.2 (Specification of evaluation order)

$$\begin{array}{lll} \textit{Redices} & R ::= & (\lambda x{:}A.\, M)\, V \mid (V_1,\, V_2).1 \mid (V_1,\, V_2).2 \mid V{<:}A \mid \\ & & \mathbf{new}_1(A)\, V \mid \mathbf{get}_1(A)\, l \mid \mathbf{set}_2(A, l)\, V \mid \\ & & \mathbf{unroll}_1(A)\, (\mathbf{roll}_2(A', V)) \\ \textit{Evaluation Contexts} & E ::= & [] \mid E\, M \mid V\, E \mid (E,\, M) \mid (V,\, E) \mid \\ & & E.1 \mid E.2 \mid E{<:}A \mid \mathbf{roll}_2(A, E) \mid \mathbf{set}_2(A, E) \end{array}$$

The evaluation order specified here is normal call by value with applications and pairs having their components evaluated from left to right.

It is straightforward to prove that values do not contain potential reduction sites and that potential reduction sites do not contain potential reduction sites as proper subterms:

Lemma [Sem] 11.7.3 $\mathit{ETV}(E) = \emptyset$

Lemma [Sem] 11.7.4 $V = E[R]$ *is impossible.*

Proof: Proved by contradiction using structural induction on V. By inspecting the definitions of values and redices, we see that no value is a redice so we must have that $E \neq []$. By inspecting the definition of evaluation contexts then, it is easily seen that $E[M]$ cannot have the forms $\lambda x{:}A.\, M'$, $<A>$, l, $\mathbf{new}_1(A)$, $\mathbf{get}_1(A)$, or $\mathbf{unroll}_1(A)$. Thus, we need worry about values with only the forms $(V_1,\, V_2)$, $\mathbf{roll}_2(A, V)$, and $\mathbf{set}_2(A, l) \Rightarrow E[R] = E_1[E_2[R]]$ where E_1 is one of $[]\, V_2$, $V_1\, []$, $\mathbf{roll}_2(A, [])$, or $\mathbf{set}_2(A, [])$. $\Rightarrow E_2[R] = V_j$ for $j \in \{1, 2\}$. But, this is impossible by the induction hypothesis so we are done. □

Lemma [Sem] 11.7.5 *If* $R = E[R']$ *then* $R = R'$ *and* $E = []$.

Proof: Inspection of the definition of redices reveals that if $R = E[R']$, $E \neq []$, then $\exists E_1, E_2.\ E[R'] = E_1[E_2[R]]$ and $E_2[R']$ is a value. By Lemma 11.7.4, it cannot be the case that $E_2[R']$ is a value so E must equal $[]$ and hence $R = R'$. □

The following lemmas relate the minimal type of a function in an application to the argument and result types of the application:

Lemma [Sem] 11.7.6 *If* $, \vdash_{\Phi} M_1\, M_2 : A$ *then* $\exists A_1, A_2.\ , \vdash_{\Phi} M_1 : A_2 \rightarrow A_1$, $, \vdash_{\Phi} M_2 : A_2$, *and* $, \vdash A_1 \leq A$.

Lemma [Sem] 11.7.7 *Suppose* $, \vdash_{\Phi} M_1\, M_2 : A$.

1. *If* $\Pi x{:}A_1'.\, A_2' \in Min_{, ,\Phi}(M_1)$ *then* $, \vdash_{\Phi} M_2 : A_1'$.

2. *If* $A_1' \rightarrow A_2' \in Min_{, ,\Phi}(M_1)$ *then* $, \vdash A_2' \leq A$.

Proof: By Lemma 11.7.6, $\exists A_1, A_2.\ , \vdash_{\Phi} M_1 : A_2 \rightarrow A_1$, $, \vdash_{\Phi} M_2 : A_2$, and $, \vdash A_1 \leq A$ $\Rightarrow\ , \vdash \Pi x{:}A_1'.\, A_2' \leq A_2 \rightarrow A_1$ (part one) or $, \vdash A_1' \rightarrow A_2' \leq A_2 \rightarrow A_1$ (part two) (definition of minimal type) $\Rightarrow\ , \vdash A_2 \leq A_1'$ (Lemma 10.5.1) $\Rightarrow\ , \vdash_{\Phi} M_2 : A_1'$ (SM-SUMP).

For part two, by Lemma 10.5.1, $, , x{:}A_2 \vdash A_2' \leq A_1$ where $x \notin \mathrm{FTV}(A_2') \cup \mathrm{FTV}(A_1)$ $\Rightarrow\ , \vdash A_2' \leq A_1$ (Theorem 10.2.4) $\Rightarrow\ , \vdash A_2' \leq A$ (S-TRAN). □

By using these results and the results on the properties of values from Section 11.5, I can prove that well-typed applications of values are always redices:

Lemma [Sem] 11.7.8 *If* $, \vdash_{\Phi} V_1\, V_2 : A$ *then* $V_1\, V_2$ *is a redice.*

Proof: By Lemma 11.7.6, $\exists A_1, A_2.\ , \vdash_{\Phi} V_1 : A_2 \rightarrow A_1 \Rightarrow V_1$ must have one of the forms: $\mathbf{new}_1(A)$, $\mathbf{get}_1(A)$, $\mathbf{set}_2(A, l)$, $\mathbf{unroll}_1(A)$, or $\lambda x{:}A'.\, V'$ (Lemma 11.5.3 and Lemma 11.5.4). Example cases:

$\lambda x{:}A'.\, V'$: Here $V_1\, V_2 = (\lambda x{:}A'.\, V')\, V_2$, a redice.

$\mathbf{get}_1(A)$: Here $V_1\, V_2 = \mathbf{get}_1(A)\, V_2$. $\Rightarrow\ , \vdash_{\Phi} V_2 : \mathbf{ref}\, A$ (Lemma 11.5.2 and Lemma 11.7.7) $\Rightarrow V_2$ is a label (Lemma 11.5.3 and Lemma 11.5.4) $\Rightarrow V_1\, V_2$ is a redice.

$\mathbf{unroll}_1(A)$: Here $V_1\, V_2 = \mathbf{unroll}_1(A)\, V_2$. $\Rightarrow\ , \vdash_{\Phi} V_2 : \mathbf{rec}\, A$ (Lemma 11.5.2 and Lemma 11.7.7) $\Rightarrow V_2$ has form $\mathbf{roll}_2(A', V')$ (Lemma 11.5.3 and Lemma 11.5.4) $\Rightarrow V_1\, V_2$ is a redice.

□

I can now prove the desired theorem about expressing closed well-typed terms in terms of values, redices, and evaluation contexts:

Theorem [Sem] 11.7.9 (Progress)
If $\bullet \vdash_{\Phi} M : A$ *then either* M *is a value or there exists an unique evaluation context* E *and an unique redice* R *such that* $M = E[R]$.

Proof: Proved by structural induction on M. By Lemma 11.7.4, it suffices to show that if M is not a value then $\exists$ an unique E and an unique R such that $M = E[R]$. Since Lemma 11.7.5 handles the cases where M is a redice, we need only handle here the cases where M is not a redice. Example remaining cases:

VAR: Here $M = x \Rightarrow \bullet \vdash_{\Phi} x : A$ which is impossible so this case cannot occur.

FST: Here $M = M'.1$, M' not a value. By Lemma 11.7.1 and the induction hypothesis then, $\exists$ an unique evaluation context E' and an unique redice R' such that $M' = E'[R'] \Rightarrow M$ can only be constructed from the evaluation context, redice pair $(E'.1)[R']$.

PAIR I: Here $M = (M_1,\ M_2)$, M_1 not a value. By Lemma 11.7.1 and the induction hypothesis, $\exists$ an unique evaluation context E' and an unique redice R' such that $M_1 = E'[R'] \Rightarrow M$ can only be constructed from the evaluation context, redice pair $((E',\ M_2))[R']$.

PAIR II: Here $M = (V,\ M')$, M' not a value. By Lemma 11.7.1 and the induction hypothesis, $\exists$ an unique evaluation context E' and an unique redice R' such that $M' = E'[R']$ $\Rightarrow M$ can only be constructed from the evaluation context, redice pair $((V,\ E'))[R']$ (Lemma 11.7.4).

APP I: Here $M = M_1\, M_2$, M_1 not a value. By Lemma 11.7.1 and the induction hypothesis, $\exists$ an unique evaluation context E' and an unique redice R' such that $M_1 = E'[R']$ $\Rightarrow M$ can only be constructed from the evaluation context, redice pair $(E'\, M_2)[R']$.

APP II: Here $M = V\, M'$, M' not a value. By Lemma 11.7.1 and the induction hypothesis, $\exists$ an unique evaluation context E' and an unique redice R' such that $M' = E'[R']$ $\Rightarrow M$ can only be constructed from the evaluation context, redice pair $(V\, E')[R']$ (Lemma 11.7.4).

APP III: Here $M = V_1\, V_2$, M not a redice. But, this is impossible by Lemma 11.7.8, so this case cannot happen.

□

The set of reductions for the semantic system follow. Note that the reductions are defined to act on term, store pairs in order to allow reductions to have side effects on the store.

Definition [Sem] 11.7.10 (Reduction on terms)

$$\begin{array}{rcl}
E[(\lambda x{:}A.\, M)\, V], S & \hookrightarrow & E[[V/x]M], S \\
E[(V_1,\, V_2).1], S & \hookrightarrow & E[V_1], S \\
E[(V_1,\, V_2).2], S & \hookrightarrow & E[V_2], S \\
E[V{<:}A], S & \hookrightarrow & E[V], S \\
\\
E[\mathbf{unroll}_1(A)\, (\mathbf{roll}_2(A', V))], S & \hookrightarrow & E[V], S \\
E[\mathbf{new}_1(A)\, V], S & \hookrightarrow & E[l], (S, l{=}V) \quad (l \notin dom(S)) \\
E[\mathbf{get}_1(A)\, l], S & \hookrightarrow & E[S(l)], S \\
E[\mathbf{set}_2(A, l)\, V], S & \hookrightarrow & E[V], [l = V]S
\end{array}$$

The first reduction rule handles the application of user defined functions via value substitution. The second and third rules handle selecting from a pair in the standard manner. The fourth rule handles coercions by dropping them. Because of the implicit subsumption rule (SM-SUMP), this action can leave the term's type unchanged.

The fifth rule handles the $\mathbf{unroll}_1(A)$ function which strips off a $\mathbf{roll}_2(A', \Leftrightarrow)$ from its argument. The remaining three rules handle the primitives dealing with references: $\mathbf{new}_1(A)$ extends the store with a new location set to its argument then returns the new location, $\mathbf{get}_1(A)$ returns the value of the location specified by its argument in the store, and $\mathbf{set}_2(A, l)$ sets the location l in the store to its argument then returns its argument. The last reduction rule uses the following function to change the store:

Definition [Sem] 11.7.11 (Setting stores)

$$\begin{array}{rcl}
[l = V]\bullet & = & \bullet \\
[l = V](S, l{=}V') & = & ([l = V]S), l{=}V \\
[l = V](S, l'{=}V') & = & ([l = V]S), l'{=}V' \quad (l' \neq l)
\end{array}$$

Because I shall show that evaluation chooses which reduction to do next deterministically, these rules mean that evaluation is completely deterministic aside from the choice of which new location name to use when extending the store. Because evaluation and typing only compare location identity, this indeterminism cannot otherwise effect the outcome of evaluation. If a completely deterministic evaluation function was desired, a notion of equivalence on terms modulo location-name conversion could be defined, or alternatively, locations could be allocated in some predefined deterministic manner.

Proving that extending and changing stores preserves the well-typedness of the store is straightforward:

Lemma [Sem] 11.7.12 *If* $\vdash S : \Phi$, $\bullet \vdash_\Phi V : A$, *and* $l \notin dom(S)$ *then* $\vdash S, l{=}V : \Phi, l{:}A$.

Proof: Inspection of the definition of $(\Phi, l{:}A)(l')$ shows that it equals A if $l' = l$ and $\Phi(l')$ otherwise. By Lemma 11.4.14, $\mathrm{dom}(S) = \mathrm{dom}(\Phi) \Rightarrow l \notin \mathrm{dom}(\Phi)$. Hence, inspection of the definition of $(\Phi, l{:}A)(l')$ shows that it equals A if $l' = l$ and $\Phi(l')$ otherwise. By STORE, $\vdash \Phi$ valid, $|S| = |\Phi|$, and $\forall l \in \mathrm{dom}(\Phi).\ \bullet \vdash_{\Phi} S(l) : \Phi(l) \Rightarrow |S, l{=}V| = |\Phi, l{:}A|$, $\vdash \Phi, l{:}A$ valid (Theorem 11.4.15 and ST-EXTEND), and $\forall l \in \mathrm{dom}(\Phi, l{:}A)$.
$\bullet \vdash_{\Phi} (S, l{=}V)(l') : (\Phi, l{:}A)(l') \Rightarrow \forall l \in \mathrm{dom}(\Phi, l{:}A)$.
$\bullet \vdash_{\Phi, l:A} (S, l{=}V)(l') : (\Phi, l{:}A)(l')$ (Theorem 11.4.18) $\Rightarrow \vdash S, l{=}V : \Phi, l{:}A$
(STORE). □

Lemma [Sem] 11.7.13 *If* $\vdash S : \Phi$ *and* $\bullet \vdash_{\Phi} V : \Phi(l)$ *then* $\vdash [l = V]S : \Phi$.

Proof: Inspection of the definition of $[l = V]S$ shows that $|[l = V]S| = |S|$ and for $l' \in \mathrm{dom}(S)$, $([l = V]S)(l') = V$ if $l' = l$ and $S(l')$ otherwise. By STORE, $\vdash \Phi$ valid, $|S| = |\Phi|$, and $\forall l \in \mathrm{dom}(\Phi).\ \bullet \vdash_{\Phi} S(l) : \Phi(l) \Rightarrow \forall l \in \mathrm{dom}(\Phi).\ \bullet \vdash_{\Phi} ([l = V]S)(l) : \Phi(l)$
$\Rightarrow \vdash [l = V]S : \Phi$ (STORE). □

The crucial theorem that reduction preserves well-typedness is then as follows:

Theorem [Sem] 11.7.14 (Subject reduction)
Suppose $\bullet \vdash_{\Phi} E[R] : A$ *and* $\vdash S : \Phi$. *Then* $\exists M', S', \Phi'$ *such that:*

1. $E[R], S \hookrightarrow E[M'], S'$
2. $\vdash S' : \Phi; \Phi'$
3. $\bullet \vdash_{\Phi;\Phi'} E[M'] : A$

Proof: By Lemma 11.7.1 and Lemma 11.7.3, $\exists A'.\ \bullet \vdash_{\Phi} R : A'$ by a rule other than SM-SUMP and if $\bullet \vdash_{\Phi;\Phi'} M' : A'$ then $\bullet \vdash_{\Phi;\Phi'} E[M'] : A$. $\Rightarrow \mathrm{FTV}(A') = \emptyset$ (Lemma 11.4.16). Casing on R:

β: Here $R = (\lambda x{:}A_1.\, M)\, V$. $\Rightarrow \exists A_2.\ \bullet \vdash_{\Phi} \lambda x{:}A_1.\, M : A_2 {\to} A'$ and $\bullet \vdash_{\Phi} V : A_2$ (SM-APP) $\Rightarrow \exists A''.\ \bullet \vdash_{\Phi} \lambda x{:}A_1.\, M : \Pi x{:}A_1.\, A''$ by a rule other than SM-SUMP and $\bullet \vdash \Pi x{:}A_1.\, A'' \leq A_2 {\to} A'$ (inspection of typing rules)
$\Rightarrow \bullet, x{:}A_1 \vdash_{\Phi} M : A''$ (SM-LAM), $\bullet \vdash A_2 \leq A_1$ and $\bullet, x{:}A_2 \vdash A'' \leq A'$
(Lemma 10.5.1) $\Rightarrow \bullet, x{:}A_2 \vdash_{\Phi} M : A''$ (Theorem 11.4.20)
$\Rightarrow \bullet, x{:}A_2 \vdash_{\Phi} M : A'$ (SM-SUMP) $\Rightarrow \bullet \vdash_{\Phi} [V/x]M : [V/x]A'$
(Corollary 11.6.13) $\Rightarrow \bullet \vdash_{\Phi} [V/x]M : A'$ (Lemma 11.6.9). Thus, it suffices to let $M' = [V/x]M$, $S' = S$, and $\Phi' = \bullet$.

Fst: Here $R = (V_1, V_2).1$. $\Rightarrow \exists x, A_2.\ \bullet \vdash_\Phi (V_1, V_2) : \Sigma x{:}A'.\, A_2$ (SM-FST) $\Rightarrow \exists A_1', A_2'.$ $\bullet \vdash_\Phi (V_1, V_2) : (A_1', A_2')$ by a rule other than SM-SUMP and $\bullet \vdash (A_1', A_2') \leq \Sigma x{:}A'.\, A_2$ (inspection of typing rules) $\Rightarrow \bullet \vdash_\Phi V_1 : A_1'$ (SM-PAIR) and $\bullet \vdash A_1' \leq A'$ (Lemma 10.5.1) $\Rightarrow \bullet \vdash_\Phi V_1 : A'$ (SM-SUMP) Thus, it suffices to let $M' = V_1$, $S' = S$, and $\Phi' = \bullet$.

Snd: Here $R = (V_1, V_2).2$. $\Rightarrow \exists A_1.\ \bullet \vdash_\Phi (V_1, V_2) : (A_1, A')$ (SM-SND) $\Rightarrow \exists A_1', A_2'.$ $\bullet \vdash_\Phi (V_1, V_2) : (A_1', A_2')$ by a rule other than SM-SUMP and $\bullet \vdash (A_1', A_2') \leq (A_1, A')$ (inspection of typing rules) $\Rightarrow \bullet \vdash_\Phi V_2 : A_2'$ (SM-PAIR) and $\bullet, x{:}A_1' \vdash A_2' \leq A'$ where $x \notin \mathrm{FTV}(A_2') \cup \mathrm{FTV}(A')$ (Lemma 10.5.1) $\Rightarrow \bullet \vdash A_2' \leq A'$ (Theorem 10.2.4) $\Rightarrow \bullet \vdash_\Phi V_2 : A'$ (SM-SUMP) Thus, it suffices to let $M' = V_2$, $S' = S$, and $\Phi' = \bullet$.

Coerce: Here $R = V {<:} A' \Rightarrow \bullet \vdash_\Phi V : A'$ (SM-COERCE) $\Rightarrow \bullet \vdash_\Phi E[V] : A$. Thus, it suffices to let $M' = V$, $S' = S$, and $\Phi' = \bullet$.

New: Here $R = \mathbf{new}_1(A_1)\, V$. By Lemmas 11.7.6, 11.5.2, and 11.7.7, $\bullet \vdash_\Phi V : A_1$ and $\bullet \vdash \mathbf{ref}\, A_1 \leq A'$. Let $M' = l$, $S' = S, l{=}V$, and $\Phi' = \bullet, l{:}A_1$ where $l \notin \mathrm{dom}(S)$. $\Rightarrow$ $\vdash S' : \Phi; \Phi'$ (Lemma 11.7.12) $\Rightarrow \bullet \vdash_{\Phi;\Phi'} l : \mathbf{ref}((\Phi; \Phi')(l))$ (SM-LABEL, EMPTY, and STORE) $\Rightarrow \bullet \vdash_{\Phi;\Phi'} l : \mathbf{ref}\, A_1$ (STORE and Lemma 11.4.14) $\Rightarrow \bullet \vdash_{\Phi;\Phi'} l : A'$ (SM-SUMP)

Get: Here $R = \mathbf{get}_1(A_1)\, l$. By Lemmas 11.7.6, 11.5.2, and 11.7.7, $\bullet \vdash_\Phi l : \mathbf{ref}\, A_1$ and $\bullet \vdash A_1 \leq A' \Rightarrow \exists A_1'.\ \bullet \vdash_\Phi l : \mathbf{ref}\, A_1'$ via a rule other than SM-SUMP and $\bullet \vdash \mathbf{ref}\, A_1' \leq \mathbf{ref}\, A_1$ (inspection of typing rules) $\Rightarrow A_1' = \Phi(l)$ (SM-LABEL) and $\bullet \vdash \mathbf{ref}\, A_1' = \mathbf{ref}\, A_1 :: \Omega$ (Lemma 10.2.2, E-REFL, and Corollary 10.3.14) $\Rightarrow \bullet \vdash_\Phi S(l) : A_1'$ (STORE) and $\bullet \vdash A_1' = A_1 :: \Omega$ (Lemma 9.4.10) $\Rightarrow \bullet \vdash_\Phi S(l) : A'$ (S-EQ, S-TRAN, and SM-SUMP) Thus, it suffices to let $M' = S(l)$, $S' = S$, and $\Phi' = \bullet$.

Set: Here $R = \mathbf{set}_2(A_1, l)\, V$. $\Rightarrow \bullet \vdash_\Phi l : \mathbf{ref}\, A_1$ (SM-SET) $\Rightarrow \exists A_1'.\ \bullet \vdash_\Phi l : \mathbf{ref}\, A_1'$ via a rule other than SM-SUMP and $\bullet \vdash \mathbf{ref}\, A_1' \leq \mathbf{ref}\, A_1$ (inspection of typing rules) $\Rightarrow A_1' = \Phi(l)$ (SM-LABEL) and $\bullet \vdash \mathbf{ref}\, A_1' = \mathbf{ref}\, A_1 :: \Omega$ (Lemma 10.2.2, E-REFL, and Corollary 10.3.14) $\Rightarrow \bullet \vdash \Phi(l) = A_1 :: \Omega$ (Lemma 9.4.10). By Lemmas 11.7.6, 11.5.2, and 11.7.7, $\bullet \vdash_\Phi V : A_1$ and $\bullet \vdash A_1 \leq A' \Rightarrow \bullet \vdash_\Phi V : A'$ (SM-SUMP) and $\bullet \vdash_\Phi V : \Phi(l)$ (E-SYM, S-EQ, and E-TRAN) $\Rightarrow \vdash [l = V]S : \Phi$ (Lemma 11.7.13). Thus, it suffices to let $M' = V$, $S' = [l = V]S$, and $\Phi' = \bullet$.

Unroll: Here $R = \mathbf{unroll}_1(A_1)\, (\mathbf{roll}_2(A_2, V))$. By Lemmas 11.7.6, 11.5.2, and 11.7.7, $\bullet \vdash_\Phi \mathbf{roll}_2(A_2, V) : \mathbf{rec}\, A_1$ and $\bullet \vdash A_1\, (\mathbf{rec}\, A_1) \leq A'$ $\Rightarrow \bullet \vdash_\Phi \mathbf{roll}_2(A_2, V) : \mathbf{rec}\, A_2$ via a rule other than SM-SUMP and

$\bullet \vdash \mathbf{rec}\, A_2 \leq \mathbf{rec}\, A_1$ (inspection of typing rules) $\Rightarrow \bullet \vdash_{\Phi} V : A_2\,(\mathbf{rec}\, A_2)$ (SM-ROLL) and $\bullet \vdash \mathbf{rec}\, A_2 = \mathbf{rec}\, A_1 :: \Omega$ (Lemma 10.2.2, E-REFL, and Corollary 10.3.14) $\Rightarrow$ $\bullet \vdash A_2 = A_1 :: \Omega{\Rightarrow}\Omega$ (Lemma 9.4.10).
$\Rightarrow \bullet \vdash A_2\,(\mathbf{rec}\, A_2) \leq A_1\,(\mathbf{rec}\, A_1)$ (C-REC, E-REFL, E-APP, and S-EQ)
$\Rightarrow \bullet \vdash_{\Phi} V : A_1\,(\mathbf{rec}\, A_1)$ (SM-SUMP) $\Rightarrow \bullet \vdash_{\Phi} V : A'$ (SM-SUMP). Thus, it suffices to let $M' = V$, $S' = S$, and $\Phi' = \bullet$.

$\square$

Thus, I have given a definition of evaluation for the semantic system that is well-defined, deterministic (modulo exact location names), and for which the semantic type system is sound:

Corollary [Sem] 11.7.15 (Soundness)
Suppose $\bullet \vdash_{\Phi} M : A$ *and* $\vdash S : \Phi$. *Then:*

1. *If* M *is a value then* M *evaluates using* S *only to* M.
2. *If* M *is not a value then* $\exists M', S', \Phi'$ *such that:*
   - $M, S \hookrightarrow M', S'$
   - *If* $M, S \hookrightarrow M'', S''$ *then* $M' = M''$ *and* $S' = S''$ *(modulo the choice of exact location names)*
   - $\vdash S' : \Phi; \Phi'$ *and* $\bullet \vdash_{\Phi;\Phi'} M' : A$

Proof: Follows from Theorems 11.7.9 and 11.7.14 and the definition of the reduction rules. $\square$

Chapter 12

Type Checking

In this chapter I consider the problem of type checking the kernel system. After discussing the difficulties with directly type checking the kernel system, I introduce a more restrictive type system and show that type checking for it is semi-decidable; in the presence of an oracle for subtyping, I show that type checking for it is decidable.

12.1 Type-Checking Difficulties

There are three difficulties with type checking the kernel system. The first difficulty is the question of how to handle the T-VAR rule, which is not syntax directed: The type checker must decide what type to apply the self function to. By Corollary 11.3.12, it is always safe to choose a semi-canonical type equal to the variable in question's type: Any other typing can be obtained from this one by the use of the T-SUMP rule. By Theorem 9.5.10, such a type is recursively computable. Thus, this difficulty is not a problem in practice.

The second difficulty is caused by the presence of the T-SUMP rule. Because the T-SUMP rule allows introducing subtyping anywhere, the type checker must be able to solve systems of inequalities rather than the usual systems of equalities handled by unification. Solving such systems is likely to be hard, especially given that even checking for subtyping between known types is undecidable.

Even worse is the third difficulty: The system does not possess minimal or "principle" types. In particular, the T-APP rule would require computing the minimal supertype of a given type that is x free (because of the non-dependent type restriction). For example, suppose , $\vdash M_2 : A_1$ and , $\vdash \lambda x{:}A_1.\,(M{<:}A_2) : \Pi x{:}A_1.\,A_2$. Then , $\vdash (\lambda x{:}A_1.\,(M{<:}A_2))\,(M_2{<:}A_1) : A$ iff , $, x{:}A_1 \vdash A_2 \leq A$ and $x \notin \mathrm{FTV}(A)$.

Such types do not exist in general, however. For example, $\Pi x'{:}{<}\Omega{\Rightarrow}\Omega{>}.\,{<}{=}x'!\,(x){::}\Omega{>}$ has two x-free supertypes, $\Pi x'{:}{<}\Omega{\Rightarrow}\Omega{>}.\,{<}\Omega{>}$ and

$\Pi x'{:}{<}{=}\lambda\alpha{::}\Omega.\,{<}\Omega{>}{::}\Omega{\Rightarrow}\Omega{>}.\,{<}{=}{<}\Omega{>}{::}\Omega{>}$, which are incomparable with each other. Hence, the kernel system in general does not possess minimal types.

Because of these difficulties, the problem of constructing a type checking procedure for the full kernel system is very hard. However, as I shall show shortly, by restricting the type system slightly, a reasonable type checking procedure can be obtained without losing any of the benefits of the translucent sum approach. Accordingly, I have chosen to leave the problem of typing checking the full kernel system to future work.

12.2 The Inference System

In this section I introduce the *inference system*, which is identical to the kernel system except for having a more restrictive type system. The major restriction is implemented by the replacement of the T-SUMP and T-COERCE rules by the new rules I-UN and I-COERCE respectively:

Definition [Inf] 12.2.1 (Replacement rules I)

$$\frac{,\ \vdash M : A' \qquad ,\ \vdash A' = A :: \Omega}{,\ \vdash M : A} \tag{I-UN}$$

$$\frac{,\ \vdash M : A' \qquad ,\ \vdash A' \leq A}{,\ \vdash M{<}{:}A : A} \tag{I-COERCE}$$

These rules specify that equality may be used anywhere (I-UN) but subtyping (subsumption) may be used only where and how specified by the programmer (I-COERCE). This restriction removes the need to handle systems of inequalities.

The inference system is not quite as restrictive as implied by the above rules: there is one point where enough information exists to make it possible for the type checker to automatically insert an explicit coercion. This point is at function application time: because we will have minimal types (in the inference system) for the function and its argument, we can insert an explicit coercion that specializes the function's type so that it operates on arguments of the actual argument type. For example, if f has minimal type $\Pi x{:}A_1.\,A_2$ and M has minimal type A_1', then we can insert an explicit coercion into the application $f\,M$ to get $(f{<}{:}\Pi x{:}A_1'.\,A_2)\,M$.

This idea is incorporated into the new I-APP rule below which replaces the old T-APP rule. The new I-VAR rule (also below), which replaces the old T-VAR rule, handles the difficulty with type checking term variables as described in the previous section. The CAN function used in the I-VAR rule is used to compute semi-canonical types; it is defined in the next section (See Lemma 12.3.1).

Definition [Inf] 12.2.2 (Replacement rules II)

$$\frac{, \vdash M_1 : \Pi x{:}A_0.\, A_1 \qquad , \vdash M_2 : A_2 \qquad , \vdash A_2 \leq A_0 \qquad , \vdash \Pi x{:}A_2.\, A_1 = A_2 \rightarrow A :: \Omega}{, \vdash M_1\, M_2 : A} \quad \text{(I-APP)}$$

$$\frac{\vdash ,\ \ \mathit{valid} \qquad x \in \mathit{dom}(,\,)}{, \vdash x : [\mathit{self}{=}x]\mathit{CAN}(,\, ,\, (x))} \quad \text{(I-VAR)}$$

These four rule replacements are the only differences between the kernel and inference systems.

Because the inference system's type system is a restriction of the kernel system's type system, all well-typed inference system terms are also well-typed in the kernel system:

Lemma [Inf] 12.2.3 *If* $, \vdash M : A$ *in the inference system then* $, \vdash M : A$ *in the kernel system.*

Proof: Proved by structural induction on derivation of $, \vdash M : A$ in the inference system. Interesting cases:

I-VAR: Here, $, \vdash x : [\text{self}{=}x]\mathit{CAN}(,\, ,\, (x))$ derived via rule I-VAR from $\vdash ,$ valid and $x \in \text{dom}(,\,) \Rightarrow\, , \vdash , (x) :: \Omega$ (Lemma 9.4.2) $\Rightarrow \mathit{CAN}(,\, ,\, (x))$ exists, $\mathit{CAN}(,\, ,\, (x)) \in C_,$, and $, \vdash , (x) = \mathit{CAN}(,\, ,\, (x)) :: \Omega$ (Lemma 12.3.1) $\Rightarrow\, , \vdash x : [\text{self}{=}x]\mathit{CAN}(,\, ,\, (x))$ in the kernel system (T-VAR).

I-APP: Here, $, \vdash M_1\, M_2 : A$ derived via rule I-APP from $, \vdash M_1 : \Pi x{:}A_0.\, A_1$, $, \vdash M_2 : A_2$, $, \vdash A_2 \leq A_0$, and $, \vdash \Pi x{:}A_2.\, A_1 = A_2 \rightarrow A :: \Omega \Rightarrow\, , \vdash M_1 : \Pi x{:}A_0.\, A_1$ and $, \vdash M_2 : A_2$ in the kernel system (induction hypothesis) and $, \vdash \Pi x{:}A_0.\, A_1 \leq \Pi x{:}A_2.\, A_1$ (Lemma 10.5.2) $\Rightarrow\, , \vdash M_1 : A_2 \rightarrow A$ in the kernel system (T-SUMP and S-EQ) $\Rightarrow\, , \vdash M_1\, M_2 : A$ in the kernel system (T-APP).

□

Note that because of this fact, the inference type system is also sound for the kernel-system semantics of Chapter 11.

The reverse result does not hold: there are well-typed kernel system terms that cannot be typed under the inference system. However, all such terms can be typed under the inference system if we insert explicit coercions where T-SUMP and T-VAR were used in the kernel system derivation:

Lemma 12.2.4 *If* $, \vdash M : A$ *in the kernel system then* $\exists M'$ *such that* M *and* M' *are the same if explicit coercions are removed and* $, \vdash M' : A$ *holds in the inference system.*

Proof: Proved by structural induction on derivation of $, \vdash M : A$ in the kernel system. Interesting cases:

T-SUMP: Here $, \vdash M : A$ derived via rule T-SUMP from $, \vdash M : A'$ and $, \vdash A' \leq A \Rightarrow \exists M''$. M and M'' are the same if explicit coercions are removed and $, \vdash M'' : A'$ in the inference system (induction hypothesis) $\Rightarrow , \vdash M''{<:}A : A$ in the inference system (I-COERCE). Hence, let $M' = M''{<:}A$ and we are done.

T-VAR: Here, $, \vdash x : [\text{self}{=}x]A$, where $, = ,_1, x{:}A_0; ,_2$, $x \notin \text{dom}(,_1)$, derived via rule T-VAR from $, \vdash A_0 = A :: \Omega$
$\Rightarrow \vdash ,$ valid and $,_1 \vdash A_0 :: \Omega$ (Lemma 6.3.5 and DECL-T), $, \vdash A_0 :: \Omega$ (Lemma 6.3.5 and Lemma 9.4.2), and $, \vdash [\text{self}{=}x]A :: \Omega$ (Theorem 11.4.15)
$\Rightarrow CAN(,_1, A_0)$ exists, $CAN(,_1, A_0) \in C_{,_1}$, $,_1 \vdash A_0 = CAN(,_1, A_0) :: \Omega$,
$CAN(, , A_0)$ exists, $CAN(, , A_0) \in C_{,}$, and $, \vdash A_0 = CAN(, , A_0) :: \Omega$ (Lemma 12.3.1)
$\Rightarrow \vdash ,_1, x{:}CAN(,_1, A_0); ,_2$ valid, $,_1, x{:}CAN(,_1, A_0); ,_2 \vdash [\text{self}{=}x]A :: \Omega$, and
$,_1, x{:}CAN(,_1, A_0); ,_2 \vdash A_0 = A :: \Omega$ (Theorem 9.4.6) and $CAN(,_1, A_0) \in C_{,}$
(Lemma 9.5.5) $\Rightarrow ,_1, x{:}CAN(,_1, A_0); ,_2 \vdash A_0 = CAN(,_1, A_0) :: \Omega$ (Theorem 9.4.1)
$\Rightarrow ,_1, x{:}CAN(,_1, A_0); ,_2 \vdash CAN(,_1, A_0) \leq A$ (E-SYM, E-TRAN, and S-EQ) $\Rightarrow$
$,_1, x{:}CAN(,_1, A_0); ,_2 \vdash [\text{self}{=}x]CAN(,_1, A_0) \leq [\text{self}{=}x]A$
(Corollary 11.3.12) $\Rightarrow , \vdash [\text{self}{=}x]CAN(,_1, A_0) \leq [\text{self}{=}x]A$
(Lemma 10.2.5).

By Theorem 9.4.6, E-SYM, and E-TRAN,
$, \vdash CAN(,_1, A_0) = CAN(, , A_0) :: \Omega \Rightarrow CAN(,_1, A_0) = CAN(, , A_0)$
(Lemma 9.5.12) $\Rightarrow , \vdash [\text{self}{=}x]CAN(, , A_0) \leq [\text{self}{=}x]A$.

By I-VAR, $, \vdash x : [\text{self}{=}x]CAN(, , A_0)$ in the inference system
$\Rightarrow , \vdash (x{<:}[\text{self}{=}x]A) : [\text{self}{=}x]A$ in the inference system (I-COERCE).

□

Thus, type checking the full kernel system can be viewed as doing type inference to determine how and where to add explicit coercions so that the terms type check in the more explicit inference system. I have chosen to treat the less explicit kernel system as the primary system rather than the inference system because of the kernel system's greater elegance and superior properties modulo deciding type checking (e.g., the inference system does not possess a replacement by a subtype property at the term level).

The requirement that the programmer specify where subsumption should be used with the exception of specializing function argument types is not likely to be too burdensome. Except when taking advantage of translucent sum abilities (e.g., to get data abstraction), such specifications are not needed. Normal computation, including polymorphic abstraction and instantiation, does not require extra specifications, thanks to the I-VAR rule. (Under the I-VAR rule, any application with an argument that has a transparent semi-canonical type does not require any extra subsumptions to type check; the only way to get a term that does not have such a type is to use an explicit coercion to make things opaque.)

Because of the restrictions made, the inference system, unlike the kernel system, possesses *principle types*. In particular, all the types possessed by a given term under , are equal under , :

Corollary [Inf] 12.2.5 (Validity of Term Types)
If $, \vdash M : A$ *then* $, \vdash A :: \Omega$*.*

Proof: Follows immediately from Lemma 12.2.3 and Theorem 11.4.15. □

Theorem [Inf] 12.2.6 (Existence of principle types)
If $, \vdash M : A_1$ *and* $, \vdash M : A_2$ *then* $, \vdash A_1 = A_2 :: \Omega$*.*

Proof: Proved by structural induction on M. WLOG, $, \vdash M : A_1$ and $, \vdash M : A_2$ were derived using rules other than I-UN (E-REFL). By Corollary 12.2.5, $, \vdash A_1 :: \Omega \Rightarrow , \vdash A_1 = A_1 :: \Omega$ (E-REFL). That takes care of the cases where $A_1 = A_2$. The interesting remaining cases are:

SND: Here $M = M'.2$, $, \vdash M' : (A_1', A_1)$, and $, \vdash M' : (A_2', A_2)$
$\Rightarrow , \vdash (A_1', A_1) = (A_2', A_2) :: \Omega$ (induction hypothesis)
$\Rightarrow , , x{:}A_1' \vdash A_1 = A_2 :: \Omega$, $x \notin \mathrm{dom}(,) \cup \mathrm{FTV}(A_1) \cup \mathrm{FTV}(A_2)$ (Lemma 9.4.10)
$\Rightarrow , \vdash A_1 = A_2 :: \Omega$ (Theorem 9.6.1).

APP: Here $M = M_1\, M_2$, $, \vdash M_1 : \Pi x{:}A_0'.\, A_1'$, $, \vdash M_2 : A_2'$, $, \vdash A_2' \leq A_0'$, $, \vdash \Pi x{:}A_2'.\, A_1' = A_2' {\to} A_1 :: \Omega$, $, \vdash M_1 : \Pi x{:}A_0''.\, A_1''$, $, \vdash M_2 : A_2''$, $, \vdash A_2'' \leq A_0''$, and $, \vdash \Pi x{:}A_2''.\, A_1'' = A_2'' {\to} A_2 :: \Omega$ where $x \notin \mathrm{dom}(,) \cup \mathrm{FTV}(A_1) \cup \mathrm{FTV}(A_2) \Rightarrow , \vdash \Pi x{:}A_0'.\, A_1' = \Pi x{:}A_0''.\, A_1'' :: \Omega$, $, \vdash A_2' = A_2'' :: \Omega$ (induction hypothesis), $, , x{:}A_2' \vdash A_1' = A_1 :: \Omega$ and $, , x{:}A_2'' \vdash A_1'' = A_2 :: \Omega$ (Lemma 9.4.10)
$\Rightarrow , , x{:}A_0' \vdash A_1' = A_1'' :: \Omega$ (Lemma 9.4.10) and
$, , x{:}A_2' \vdash A_1'' = A_2 :: \Omega$ (Theorem 9.4.6) $\Rightarrow , , x{:}A_2' \vdash A_1' = A_1'' :: \Omega$ (Theorem 10.4.7)
$\Rightarrow , , x{:}A_2' \vdash A_1 = A_2 :: \Omega$ (E-SYM and E-TRAN)
$\Rightarrow , \vdash A_1 = A_2 :: \Omega$ (Theorem 9.6.1).

□

This result means that the type checker can deal with equivalence classes of equal types rather than individual types; there is no need to determine in advance which representative of the current class to use.

```
$VALID(,) =$ if $\vdash ,$ valid then return else raise fail

$KIND(,, A) =$ if $\exists K. , \vdash A :: K$ then return($K$) else raise fail

$EQUAL(,, A_1, A_2, K) =$ if $, \vdash A_1 = A_2 :: K$ then return else raise fail

$SUB(,, A_1, A_2) =$ if $, \vdash A_1 \leq A_2$ then return else raise fail or hang


$CAN(,, A) =$ If $, \vdash A :: \Omega$ then
                 return($A'$) such that $A' \in C_{,}$ and $, \vdash A = A' :: \Omega$
               else
                 raise fail
```

Figure 12.1: Auxiliary procedures for type checking

12.3 Semi-Decidability

In this section I give a procedure that decides the well-typed term judgment for the inference system and show that it is semi-decidable (decidable given an oracle for subtyping). First, I will need some auxiliary procedures:

Lemma 12.3.1 (Existence of auxiliary procedures)
Procedures exist which have the behavior specified in Figure 12.1. VALID, KIND, EQUAL, and CAN are algorithms while SUB may fail to return only in the negative case.

Proof: Follows from Theorem 9.4.11, Corollary 9.4.12, Theorem 10.5.6, and Theorem 9.5.10. □

The decision procedure is given in Figure 12.2. $TYPE(,, M)$ computes a valid type for M under , unless M is not a well-typed term under , , in which case it fails. $TYPE$ uses CAN when it is necessary to find an equivalence-class representative having a certain form. (This method works because semi-canonical types are maximally defined and have the minimal number of dependencies.) The proof of the correctness of the $TYPE$ procedure is straightforward:

Lemma [Inf] 12.3.2 *If $TYPE(,, M)$ returns A then $, \vdash M : A$.*

```
$TYPE(, , M) = VALID(, );$
              case $M$ of
           $x$: if $x \in \mathrm{dom}(, )$ then
                  return([self=$x$]$CAN(, , , (x))$)
                else
                  raise fail
$\lambda x{:}A.\, M,\ x \notin \mathrm{dom}(, )$: return($\Pi x{:}A.\, TYPE((, , x{:}A), M)$)
     $M_1\, M_2$: case $CAN(, , TYPE(, , M_1))$ of
                  $\Pi x{:}A_0.\, A_1$: $SUB(, , TYPE(, , M_2), A_0);$
                          case $CAN(, , \Pi x{:}TYPE(, , M_2).\, A_1)$ of
                            $A \rightarrow A'$: return($A'$)
  $(M_1,\ M_2)$: return($(TYPE(, , M_1), TYPE(, , M_2))$)
       $M.1$: case $CAN(, , TYPE(, , M))$ of
                                $\Sigma x{:}A.\, A'$: return($A$)
       $M.2$: case $CAN(, , TYPE(, , M))$ of
                                $(A, A')$: return($A'$)
       $<A>$: $<=A::KIND(, , A)>$
      $M<:A$: $SUB(, , TYPE(, , M), A);$
              return($A$)
      **new**: return($\forall\alpha::\Omega.\alpha\rightarrow\mathbf{ref}\,\alpha$)
      **get**: return($\forall\alpha::\Omega.\mathbf{ref}\,\alpha\rightarrow\alpha$)
      **set**: return($\forall\alpha::\Omega.\mathbf{ref}\,\alpha\rightarrow(\alpha\rightarrow\alpha)$)
     **roll**: return($\forall\alpha::\Omega\Rightarrow\Omega.\alpha\,(\mathbf{rec}\,\alpha)\rightarrow\mathbf{rec}\,\alpha$)
   **unroll**: return($\forall\alpha::\Omega\Rightarrow\Omega.\mathbf{rec}\,\alpha\rightarrow\alpha\,(\mathbf{rec}\,\alpha)$)


$TERM(, , M, A) = EQUAL(, , TYPE(, , M), A, \Omega)$
```

Figure 12.2: Decision procedure for type checking

Proof: Proved by structural induction on M. By inspection of the code for $TYPE$, $TYPE(, , M)$ returns $\Rightarrow VALID(,)$ returns $\Rightarrow \vdash ,$ valid (Lemma 12.3.1). Example cases:

new: Here $TYPE(, , \textbf{new})$ returns $\forall\alpha{::}\Omega.\alpha{\to}\textbf{ref}\,\alpha$. By T-NEW, $, \vdash \textbf{new} : \forall\alpha{::}\Omega.\alpha{\to}\textbf{ref}\,\alpha$.

$<A>$: Here $TYPE(, , <A>)$ returns $<=A{::}KIND(, , A)> \Rightarrow KIND(, , A)$ returns $\Rightarrow$ $, \vdash A :: KIND(, , A)$ (Lemma 12.3.1)
$\Rightarrow , \vdash <A> : <=A{::}KIND(, , A)>$ (T-REIFY).

$M{<:}A$: Here $TYPE(, , M{<:}A)$ returns A and $SUB(, , TYPE(, , M), A)$ returns
$\Rightarrow TYPE(, , M)$ returns and $, \vdash TYPE(, , M) \leq A$ (Lemma 12.3.1)
$\Rightarrow , \vdash M : TYPE(, , M)$ (induction hypothesis) $\Rightarrow , \vdash M{<:}A : A$ (I-COERCE).

$M_1\,M_2$: Here $TYPE(, , M_1\,M_2)$ returns A' where $\exists x', A_0, A_1$.
$CAN(, , TYPE(, , M_1)) = \Pi x{:}A_0.\,A_1$, $SUB(, , TYPE(, , M_2), A_0)$ returns,
$CAN(, , \Pi x{:}TYPE(, , M_2).\,A_1) = A{\to}A'$, and $x \notin \text{dom}(,) \cup \text{FTV}(A')$

$\Rightarrow , \vdash TYPE(, , M_2) \leq A_0$, $, \vdash TYPE(, , M_1) = \Pi x{:}A_0.\,A_1 :: \Omega$,
$, \vdash \Pi x{:}TYPE(, , M_2).\,A_1 = A{\to}A' :: \Omega$ (Lemma 12.3.1), $TYPE(, , M_1)$ returns, and $TYPE(, , M_2)$ returns

$\Rightarrow , \vdash M_1 : TYPE(, , M_1)$, $, \vdash M_2 : TYPE(, , M_2)$ (induction hypothesis),
$, \vdash TYPE(, , M_2) = A :: \Omega$, and $, , x{:}TYPE(, , M_2) \vdash A_1 = A' :: \Omega$ (Lemma 9.4.10)
$\Rightarrow , \vdash M_1 : \Pi x{:}A_0.\,A_1$ (I-UN) and
$, \vdash \Pi x{:}TYPE(, , M_2).\,A_1 = TYPE(, , M_2){\to}A' :: \Omega$ (Theorem 9.4.3, E-REFL, and E-DFUN) $\Rightarrow , \vdash M_1\,M_2 : A'$ (I-APP).

□

Lemma [Inf] 12.3.3 *If* $, \vdash M : A$ *then* $TYPE(, , M)$ *returns.*

Proof: Proved by structural induction on $, \vdash M : A$. By Corollary 12.2.5, $, \vdash A :: \Omega$ $\Rightarrow \vdash ,$ valid (Lemma 6.3.5) $\Rightarrow VALID(,)$ returns (Lemma 12.3.1). Example cases:

I-VAR: Here $, \vdash x : [\text{self}{=}x]CAN(, , , (x))$ derived via rule I-VAR from $\vdash ,$ valid and $x \in \text{dom}(,) \Rightarrow TYPE(, , x)$ returns.

T-REIFY: Here $, \vdash <A> : <=A{::}K>$ derived via rule T-REIFY from $, \vdash A :: K \Rightarrow KIND(, , A)$ returns K (Lemma 12.3.1) $\Rightarrow TYPE(, , <A>)$ returns (inspection of TYPE code).

I-COERCE: Here $, \vdash M{<:}A : A$ derived via rule I-COERCE from $, \vdash M : A'$ and $, \vdash A' \leq A$
$\Rightarrow TYPE(,, M)$ returns (induction hypothesis)
$\Rightarrow\ , \vdash M : TYPE(,, M)$ (Lemma 12.3.2) $\Rightarrow\ , \vdash TYPE(,, M) = A' :: \Omega$ (Theorem 12.2.6) $\Rightarrow\ , \vdash TYPE(,, M) \leq A$ (S-EQ and S-TRAN)
$\Rightarrow SUB(,, TYPE(,, M), A)$ returns (Lemma 12.3.1) $\Rightarrow TYPE(,, M{<:}A)$ returns (inspection of TYPE code).

I-SND: Here $, \vdash M.2 : A_2$ derived via rule T-SND from $, \vdash M : (A_1, A_2)$
$\Rightarrow TYPE(,, M)$ returns (induction hypothesis) $\Rightarrow\ , \vdash M : TYPE(,, M)$ (Lemma 12.3.2) $\Rightarrow\ , \vdash TYPE(,, M) :: \Omega$ (Corollary 12.2.5) and
$, \vdash TYPE(,, M) = (A_1, A_2) :: \Omega$ (Theorem 12.2.6)

$\Rightarrow CAN(,, TYPE(,, M))$ returns A' such that $A' \in C_{,}$ and
$, \vdash TYPE(,, M) = A' :: \Omega$ (Lemma 12.3.1) $\Rightarrow\ , \vdash (A_1, A_2) = A' :: \Omega$ (E-SYM and E-TRAN)

$\Rightarrow \exists x, A'_1, A'_2.\ A' = \Sigma x{:}A'_1.\ A'_2$ and $x \notin \mathrm{dom}(,) \cup \mathrm{FTV}(A_2)$ (Lemma 9.5.7)

$\Rightarrow\ ,, A'_1{:} \vdash A'_2 = A_2 :: \Omega$ (Lemma 9.4.10 and E-SYM) and $A'_2 \in C_{,\,,x:A'_1}$ (Lemma 9.5.3)

$\Rightarrow x \notin \mathrm{FTV}(A'_2)$ (Corollary 9.6.3) $\Rightarrow A' = (A'_1, A'_2) \Rightarrow TYPE(,, M.2)$ returns A'_2 (inspection of TYPE).

I-APP: Here $, \vdash M_1\, M_2 : A'$ derived via rule I-APP from $, \vdash M_1 : \Pi x{:}A_0.\ A_1$, $, \vdash M_2 : A_2$, $, \vdash A_2 \leq A_0$ and $, \vdash \Pi x{:}A_2.\ A_1 = A_2 {\rightarrow} A' :: \Omega$ where $x \notin \mathrm{dom}(,) \cup \mathrm{FTV}(A') \Rightarrow$ $TYPE(,, M_1)$ and $TYPE(,, M_2)$ return (induction hypothesis) and
$,, x{:}A_2 \vdash A_1 = A' :: \Omega$ (Lemma 9.4.10) $\Rightarrow\ , \vdash M_1 : TYPE(,, M_1)$ and
$, \vdash M_2 : TYPE(,, M_2)$ (Lemma 12.3.2) $\Rightarrow\ , \vdash TYPE(,, M_1) :: \Omega$ (Corollary 12.2.5),
$, \vdash \Pi x{:}A_0.\ A_1 = TYPE(,, M_1) :: \Omega$, and $, \vdash A_2 = TYPE(,, M_2) :: \Omega$ (Theorem 12.2.6)

$\Rightarrow\ ,, x{:}TYPE(,, M_2) \vdash A_1 = A' :: \Omega$ (Theorem 9.4.6) and
$CAN(,, TYPE(,, M_1))$ returns A'' such that $A'' \in C_{,}$ and
$, \vdash TYPE(,, M_1) = A'' :: \Omega$ (Lemma 12.3.1) $\Rightarrow\ , \vdash A'' = \Pi x{:}A_0.\ A_1 :: \Omega$ (E-SYM and E-TRAN)

$\Rightarrow \exists A'_0, A'_1.\ A'' = \Pi x{:}A'_0.\ A'_1$ (Lemma 9.5.7 and E-SYM)
$\Rightarrow\ , \vdash A'_0 = A_0 :: \Omega$, $,, x{:}A'_0 \vdash A'_1 = A_1 :: \Omega$, (Lemma 9.4.10) and
$,, x{:}A'_0 \vdash A'_1 :: \Omega$ (Theorem 9.4.3 and C-DFUN) $\Rightarrow\ , \vdash TYPE(,, M_2) \leq A'_0$ (E-SYM, S-EQ, and S-TRAN) $\Rightarrow SUB(,, TYPE(,, M_2), A'_0)$ returns
(Lemma 12.3.1), $,, x{:}TYPE(,, M_2) \vdash A'_1 :: \Omega$, and $,, x{:}TYPE(,, M_2) \vdash A_1 = A'_1 :: \Omega$ (Theorem 10.4.7) $\Rightarrow\ , \vdash \Pi x{:}TYPE(,, M_2).\ A'_1 :: \Omega$ (C-DFUN) and
$,, x{:}TYPE(,, M_2) \vdash A'_1 = A' :: \Omega$ (E-TRAN)

$\Rightarrow CAN(, ,\Pi x{:}TYPE(, ,M_2).\, A'_1)$ returns A''' such that $A''' \in C_{,}$ and
$, \vdash \Pi x{:}TYPE(, ,M_2).\, A'_1 = A''' :: \Omega$ (Lemma 12.3.1)
$\Rightarrow \exists A'_8, A'_9.\ A''' = \Pi x{:}A'_8.\, A'_9$ (Lemma 9.5.7)
$\Rightarrow ,\ \vdash A'_8 = TYPE(, ,M_2) :: \Omega, , ,x{:}TYPE(, ,M_2) \vdash A'_9 = A'_1 :: \Omega$
(Lemma 9.4.10) and $A'_9 \in C_{, ,x:A'_8}$ (Lemma 9.5.3)
$\Rightarrow , ,x{:}TYPE(, ,M_2) \vdash A'_9 = A' :: \Omega$ (E-TRAN)
$\Rightarrow , ,x{:}A'_8 \vdash A'_9 = A' :: \Omega$ (Theorem 9.4.6) $\Rightarrow x \notin \mathrm{FTV}(A'_9)$ (Corollary 9.6.3) $\Rightarrow$
$A''' = A'_8 \to A'_9 \Rightarrow TYPE(, ,M_1\, M_2)$ returns A'_9 (inspection of TYPE).

□

Moreover, if we had an oracle for the SUB procedure, then the $TYPE$ procedure would always terminate:

Lemma [Inf] 12.3.4 *If the SUB procedure in $TYPE$ is replaced by an oracle for subtyping which never hangs then $TYPE(, ,M)$ always terminates.*

Proof: By Lemma 12.3.1, $VALID$, CAN, and $KIND$ always terminate. Since all recursive calls to $TYPE$ are on proper sub-terms of the original term, $TYPE(, ,M)$ can only fail to terminate if a call to SUB fails to terminate. □

Thus, because of these results and the fact that the inference system has principle types (Theorem 12.2.6), the inference system's well-typed term judgment is semi-decidable (decidable given an oracle for subtyping):

Theorem [Inf] 12.3.5 (Semi-decidability)
The judgment $,\ \vdash M : A$ is semi-decidable. Non-termination only occurs if the judgment is false. If an oracle is provided for subtyping then the judgment becomes decidable.

Proof: Claim: $TERM(, ,M,A)$ returns iff $,\ \vdash M : A$:

$\Rightarrow$) Here $TERM(, ,M,A)$ returns $\Rightarrow TYPE(, ,M)$ returns and
$,\ \vdash TYPE(, ,M) = A :: \Omega$ (Lemma 12.3.1) $\Rightarrow ,\ \vdash M : TYPE(, ,M)$
(Lemma 12.3.2) $\Rightarrow ,\ \vdash M : A$ (I-UN)

$\Leftarrow$) Here $,\ \vdash M : A \Rightarrow TYPE(, ,M)$ returns (Lemma 12.3.3)
$\Rightarrow ,\ \vdash M : TYPE(, ,M)$ (Lemma 12.3.2) $\Rightarrow ,\ \vdash TYPE(, ,M) = A :: \Omega$
(Theorem 12.2.6) $\Rightarrow EQUAL(, ,TYPE(, ,M), A, \Omega)$ (Lemma 12.3.1)
$\Rightarrow TERM(, ,M,A)$ returns

Hence, the judgment is semi-decidable, with non-termination only occuring if the judgment is false. By Lemmas 12.3.4 and 12.3.1, $TERM$ always terminates if an oracle is used for subtyping $\Rightarrow$ the judgment is decidable in the presence of an oracle for subtyping. □

Part III

Conclusions

Chapter 13

Extensions

In this chapter I discuss briefly a number of possible extensions to the basic kernel system. Some of them are straightforward while others will likely require extensive future research.

13.1 N-Ary Named Fields

One extension needed to make the kernel system a more realistic programming language is the ability to have translucent sums with any number of named fields. This extension would allow us to create translucent sum terms like **module** `a=3; x=5; y=7;` **end** with type **interface** `a:int; x:int; y:int;` **end**. Without this ability, we would have to use a more cumbersome term like $(3,(5,7))$ with type (`int`,(`int`,`int`)); we would also have to remember when accessing fields that `.a` corresponds to .1, `.x` to .2.1, and `.y` to .2.2. This requirement unnecessarily burdens the programmer's memory and makes adding or removing fields later difficult.

Because translucent sums are dependent sums, there must be a way for the types of components to refer to previous fields. Naively, one might expect that the field names could be reused for this purpose. This idea does not work because of a conflict between the requirements on external and internal names: Internal names must be free to α-vary under substitution, β-reduction, and other operators while external names must remain fixed (otherwise, the set of legal field names which can be selected on changes).

A simple solution is just to have separate external and internal names for each field. As an example, with internal names denoted in parentheses, we might have the following type:

```
interface
    T(α):<type>;
    S(S):<= interface T(β):<type>; x(x):β!->α!; end ::type>;
end
```

Note that in the absence of separate names there would be no way to express the fact that the `x` field's type depends on the types of both the first and second `T` fields. The scope of internal names is the remaining fields of the translucent sum type; external names are never in scope. In order to simplify the system and catch programmer errors, duplicate external names in a translucent sum should be prohibited.

I expect that different internal and external names will seldom be needed in practice. Accordingly, a useful abbreviation is to allow the internal name to be omitted when it is the same as the external one. Translucent sum terms in the kernel system, for simplicity, do not contain internal bindings and thus do not need internal names. This ability is easy added, however, using syntactic sugar. For example, the internal bindings in **module** `x(a)=2; y=a+a;` **end** can be removed by rewriting the term to **let val** `a=2;` **val** `y=a+a;` **in module** `x=a; y=y;` **end**.

Because of the way I factored translucent sums in the kernel system, they contain only values directly; constructors can be contained only indirectly via the use of a reified constructor. This factoring could either be hidden from the programmer by syntactic sugar or removed entirely; the result would be translucent sums that (appear) to have both constructor and value components. The previous example type would then be as follows:

```
interface
    constr T(α)::type;
    constr S = interface constr T(β)::type; val x:β->α; end;
end
```

Here constructor fields are indicated by the keyword **constr** and value fields by the keyword **val**; opaquely defined constructor fields like `T` just list the kind of the field while transparently defined constructor fields like `S` give the constructor they are defined equal to (the kind can be inferred). Note that because α and β are types here, constructor extractions ($\Leftrightarrow$!) are no longer needed. Since most constructors are types, it is convenient to allow the **type** keyword to be used as syntactic sugar for a **constr** declaration of kind **type** (e.g., **type** `T(α);` or **type** `U = int;`).

Although the factored version is simpler theoretically, it may be less efficient in practice. Consider an implementation that wishes to save memory by omitting constructor fields (the kernel system semantics do not require any type information at runtime). This optimization is easily done in the non-factored version because the two sorts of fields are syntactically distinguished.

In the factored version, on the other hand, it is impossible in general to tell what sort of field we are dealing with. In particular, a field's sort may depend on whether or not a free type variable whose identity is not known at compile time is equal to a reified constructor type. Because traditional representation methods require that a type's representation be invariant under type substitution, this optimization requires non-traditional representation methods in the factored case. New representation methods using *intensional polymorphism* [23, 44] may allow this optimization, but probably not without some runtime and compiler-complexity cost.

What notions of equality and subtyping are appropriate for n-ary named translucent sums? One approach is to just extend the kernel system rules in the obvious way. Using a factored version, this approach might look as follows:

$$\frac{\vdash ,\ \text{valid}}{,\ \vdash \{\} = \{\} :: \Omega} \qquad \text{(E-BASE)}$$

$$\frac{\begin{array}{c} ,\ \vdash A = A' :: \Omega \\ ,\ , x{:}A \vdash \{F\} = \{F'\} :: \Omega \qquad n \notin \text{fields}(F) \end{array}}{,\ \vdash \{n(x){:}A;\ F\} = \{n(x){:}A';\ F'\} :: \Omega} \qquad \text{(E-FIELD)}$$

$$\frac{\vdash ,\ \text{valid}}{,\ \vdash \{\} \leq \{\}} \qquad \text{(S-BASE)}$$

$$\frac{\begin{array}{c} ,\ \vdash A \leq A' \qquad ,\ , x{:}A' \vdash \{F'\} :: \Omega \\ ,\ , x{:}A \vdash \{F\} \leq \{F'\} \qquad n \notin \text{fields}(F) \end{array}}{,\ \vdash \{n(x){:}A;\ F\} \leq \{n(x){:}A';\ F'\}} \qquad \text{(S-FIELD)}$$

Here, F and F' represent possibly empty lists of fields; fields(F) is the set of field names defined in F. I am using { and } instead of **interface** and **end** for conciseness. Notice that these rules only allow translucent sums with the same field names in the same order to be equal to or subtypes of each other.

13.2 Extended Interface Matching

Another approach is to provide what I call *extended interface matching*, the ability to reorder or drop fields using subtyping. This ability is very useful in practice because it allows modules to be combined in more flexible ways. For example, using the dropping ability, a programmer could pass a module with more features than required to a functor without having to manually write a coercion function to extract only the required fields. Likewise, the reordering ability frees programmers of the need to remember the exact order of the fields of an interface.

Adding the following rule to the rules of the previous section permits subtyping to drop fields at the end of a translucent sum:

$$\frac{,\ \vdash \{F\} :: \Omega}{,\ \vdash \{F\} \leq \{\}} \qquad \text{(DROP-END)}$$

Dropping fields at the end is always safe because the fields that remain cannot depend on the fields being dropped. In order to drop fields in the middle, we must check that none of the later fields depend on the fields being dropped:

$$\frac{\begin{array}{c} x \notin \mathrm{FTV}(F') \\ ,\ , x{:}A \vdash \{F\} \leq \{F'\} \qquad n \notin \mathrm{fields}(F) \end{array}}{,\ \vdash \{n(x){:}A;\ F\} \leq \{F'\}} \qquad \text{(DROP-ANY)}$$

What rules to use for reordering fields is problematic. In the original formulation of the kernel system [20], rules equivalent to the following rule were used:

$$\frac{\begin{array}{c} x_1 \notin \mathrm{FTV}(A_2) \qquad x_2 \notin \mathrm{FTV}(A_1) \\ ,\ \vdash \{n_1(x_1){:}A_1;\ n_2(x_2){:}A_2;\ F\} :: \Omega \end{array}}{,\ \vdash \{n_1(x_1){:}A_1;\ n_2(x_2){:}A_2;\ F\} \leq \{n_2(x_2){:}A_2;\ n_1(x_1){:}A_1;\ F\}} \qquad \text{(SWAP)}$$

This rule allows adjacent fields be swapped if the later field does not refer to the earlier one.

I have since discovered a problem with this rule: It does not treat nested translucent sums the same as it does the equivalent flattened version. For example, under the assignment $\bullet, \beta{::}\Omega$, we have that

$$\begin{array}{rl} & \{x_a{:}{<}\Omega{>};\ x_b{:}\beta;\ y{:}{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\ \alpha\, x_a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\} \\ \leq & \{x_a{:}{<}\Omega{>};\ y{:}{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\ \alpha\, x_a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\ x_b{:}\beta;\} \\ = & \{x_a{:}{<}\Omega{>};\ y{:}{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\ \alpha\, x_a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\ x_b{:}y!\,(\lambda\alpha{::}\Omega.\,\beta);\} \\ \leq & \{x_a{:}{<}\Omega{>};\ y{:}{<}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\ x_b{:}y!\,(\lambda\alpha{::}\Omega.\,\beta);\} \\ \leq & \{y{:}{<}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\ x_a{:}{<}\Omega{>};\ x_b{:}y!\,(\lambda\alpha{::}\Omega.\,\beta);\} \end{array}$$

The subtyping fails to hold though, if we combine x_a and x_b together in a single sum nested inside the outer sum:

$$\begin{array}{rl} & \{x{:}\{a{:}{<}\Omega{>};\ b{:}\beta;\};\ y{:}{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\ \alpha\, x.a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\} \\ \leq & \{x{:}\{a{:}{<}\Omega{>};\ b{:}\beta;\};\ y{:}{<}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\} \\ \leq & \{y{:}{<}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\ x{:}\{a{:}{<}\Omega{>};\ b{:}\beta;\};\} \\ \not\leq & \{y{:}{<}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>};\ x{:}\{a{:}{<}\Omega{>};\ b{:}y!\,(\lambda\alpha{::}\Omega.\,\beta);\};\} \end{array}$$

(We are forced here to forget the information about y's identity in order to swap the x and y fields — swapping requires removing the dependency on x. Unfortunately, we need this information after the swap is done in order to relate b's type to y.)

I find this behavior counterintuitive. We could make subtyping treat nested and flattened sums the same by adding flattening rules so that, for example,

$$\{F_1; x{:}\{a{:}A_1;\ b{:}A_2;\};F_2\} \leq \{F_3; x{:}\{a{:}A_1';\ b{:}A_2';\};F_4\}$$

holds if

$$\begin{array}{l}\{F_1;\, x_a{:}A_1;x_b{:}[x_a/a]A_2;[x_a/x.a][x_b/x.b]F2\} \quad \leq \\ \{F_3;\, x_a{:}A_1';x_b{:}[x_b/b]A_2';[x_a/x.a][x_b/x.b]F4\}\end{array}$$

provided no variable capture or field duplication occurs. In the more general case, the x field on the right-hand side might have fewer subfields in a different order or might not exist at all.

The decision problem for these rules (both with and without flattening) appears to be quite difficult; it is hard to see how the need to guess intermediate types can be avoided. (E.g., what type should we coerce a field to before we swap it with the preceding field?)

An alternative way of handling reordering can be obtained by imagining translating the system with reordering and dropping of fields into the system of the previous section by inserting coercion functions into the code whenever subsumption occurs. Thus, for example, if subsumption was used to swap the fields of a translucent sum with type $\{x{:}A_1;\ y{:}A_2;\}$ resulting in the type $\{y{:}A_2';\ x{:}A_1';\}$, the following function of type $\{x{:}A_1;\ y{:}A_2;\}{\rightarrow}\{y{:}A_2';\ x{:}A_1';\}$ would be inserted[1]:

$$\lambda r{:}\{x{:}A_1;\ y{:}A_2;\}.\,(\{y{=}r.y;\ x{=}r.x;\}{<:}\{y{:}A_2';\ x{:}A_1';\})$$

To make this insertion work, the subtyping rules must ensure that this function is always well typed in the previous section's system whenever they allow $\{x{:}A_1;\ y{:}A_2;\}$ to be a subtype of $\{y{:}A_2';\ x{:}A_1';\}$.

By examining the type rules of the target system, it can be seen that the following rule is the most general rule possible for this case:

$$\frac{\begin{array}{c}, \vdash \{y(x_2){:}A_2';\ x(x_1){:}A_1';\} :: \Omega \\ ,\ , x_1{:}A_1, x_2{:}A_2 \vdash [\text{self}{=}x_1]A_1 \leq A_1' \\ ,\ , x_1{:}A_1, x_2{:}A_2 \vdash [\text{self}{=}x_2]A_2 \leq A_2'\end{array}}{, \vdash \{x(x_1){:}A_1;\ y(x_2){:}A_2;\} \leq \{y(x_2){:}A_2';\ x(x_1){:}A_1';\}} \qquad \text{(TWO-FLIP)}$$

The original SWAP rule can be derived from this rule. However, unlike the original rule, this rule treats nested and flattened translucent sums the same. The example that we needed extra flattening rules to handle before can be done using just the new TWO-FLIP rule: Under the assignment $\bullet, \beta{::}\Omega, x{:}\{a{:}{<}\Omega{>};\ b{:}\beta;\}, y{:}{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\ \alpha\, x.a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>}$,

[1]To simplify the discussion, I am assuming here that no reordering or dropping of subfields is needed on the contents of the x and y fields. In the general case it may be necessary to insert additional coercion functions before $r.y$ and $r.x$ in order to handle this.

we have that

$$\begin{array}{ll}
[\text{self}{=}x]\{a{:}{<}\Omega{>};\ b{:}\beta;\} & = \\
\{a{:}{<}{=}x.a!{::}\Omega{>};\ b{:}\beta;\} & = \\
\{a{:}{<}{=}x.a!{::}\Omega{>};\ b{:}(\lambda\alpha{::}\Omega.\,\beta)\,x.a!;\} & = \\
\{a{:}{<}{=}x.a!{::}\Omega{>};\ b{:}(\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\,\alpha\,x.a!)\,(\lambda\alpha{::}\Omega.\,\beta);\} & = \\
\{a{:}{<}{=}x.a!{::}\Omega{>};\ b{:}y!\,(\lambda\alpha{::}\Omega.\,\beta);\} & \leq \\
\{a{:}{<}\Omega{>};\ b{:}y!\,(\lambda\alpha{::}\Omega.\,\beta);\} &
\end{array}$$

and

$$\begin{array}{ll}
[\text{self}{=}y]{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\,\alpha\,x.a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>} & = \\
{<}{=}\lambda\alpha{::}\Omega{\Rightarrow}\Omega.\,\alpha\,x.a!{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>} & \leq \\
{<}(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega{>} &
\end{array}$$

Thus, TWO-FLIP handles the nested translucent sum example.

Because neither of these abilities (to derive SWAP and to handle nesting properly) require the use of the self function, a simpler and less powerful rule could be substituted for TWO-FLIP without losing either of these abilities:

$$\frac{\begin{array}{c}
,\ \vdash \{y(x_2){:}A_2';\ x(x_1){:}A_1';\} :: \Omega \\
,\ , x_1{:}A_1, x_2{:}A_2 \vdash A_1 \leq A_1' \\
,\ , x_1{:}A_1, x_2{:}A_2 \vdash A_2 \leq A_2'
\end{array}}{,\ \vdash \{x(x_1){:}A_1;\ y(x_2){:}A_2;\} \leq \{y(x_2){:}A_2';\ x(x_1){:}A_1';\}} \qquad \text{(SIMPLER)}$$

The most important difference between these rules is that TWO-FLIP, unlike SIMPLER, allows the swapping of fields that must contain identical constructors. For example, only under TWO-FLIP is $\{x{:}{<}\Omega{>};\ y{:}{<}{=}x!{::}\Omega{>};\}$ a subtype of $\{y{:}{<}\Omega{>};\ x{:}{<}{=}y!{::}\Omega{>};\}$. The use of the self function is crucial here:

$$x{:}{<}\Omega{>}, y{:}{<}{=}x!{::}\Omega{>} \vdash [\text{self}{=}x]{<}\Omega{>} = {<}{=}x!{::}\Omega{>} = {<}{=}y!{::}\Omega{>}$$

It is unclear how important this ability is in practice however.

The analysis based on the translation idea can be extended to deal with reordering and dropping an arbitrary number of fields at the one time, producing generalized versions of TWO-FLIP and SIMPLER. Xavier Leroy's system [31] (see Section 14.3) appears to be using a generalized variant of SIMPLER. The decision problem for these rules (TWO-FLIP, SIMPLER, and generalizations thereof) does not seem to be much more difficult than the current kernel-system subtyping problem.

13.3 Transparent Value Declarations

Translucent-sum types as I have described them so far allow opaque constructor declarations, transparent constructor declarations, and opaque value declarations. By analogy with opaque and transparent constructor declarations, we could consider adding *transparent value declarations* (TVDs for short) as well. A transparent value declaration would specify the existence of a value component with a given type those contents is equal to a specified term. As an example, consider the following type with TVDs:

```
interface
    val x:int = 3;
    val X:interface type T; type S; val n:int; end;
    val Y:interface type T; type S; val n:int; end = X;
end
```

If `M` is a module declared to have this interface, then we can infer that `M.x` $= 3$ and `M.X` $=$ `M.Y`; hence we can also infer that `M.X.T` $=$ `M.Y.T`, `M.X.S` $=$ `M.Y.S`, and `M.X.n` $=$ `M.Y.n`.

There are three reasons why we might want to extend translucent sums to allow transparent value declarations. First, TVDs support inlining; for example, when compiling a module that uses module `M`, the compiler can use the fact that `M.x` $= 3$ to replace references to `M.x` with 3. (This ability is, of course, more useful when `M.x` is a complicated function.)

Second, TVDs allow *value sharing*, the ability to express the fact that two modules refer to the same value. This ability may be useful when it is important that two modules constructed independently use a common piece of code. See [37] for some good examples of where value sharing is useful.

Third, TVDs allow *module sharing* (also called structure sharing); module sharing permits equating all the components of two modules with a single equation (e.g., `M.X` $=$ `M.Y`) which would otherwise require possibly exponentially many equations equating all the individual components of the modules. Module sharing is thus a useful abbreviation mechanism.

What terms should we allow TVDs to declare value components' contents to be equal to? In order to be able to express value or module sharing, we will need to allow places ($x\rho$'s). Since subject reduction (desirable for soundness) requires that the set of types be closed under value substitution, we must then also allow values and projections from values, resulting in *extended values*:

Definition 13.3.1 (Syntax)

Extended Values $\nu \quad ::= \quad x \mid V \mid \nu.n$

If we are concerned with only inlining, then allowing only values may suffice.

Either of these choices has the drawback that it greatly increases the complexity of the system: types may contain values which may contain arbitrary terms because of functional values so that the type-validity judgment must then depend on the full valid term and subtyping judgments. An actual programming language could restrict the set of legal terms in TVDs further to simplify type inference but this restriction would not change the underlying theory or its difficulty.

If we allow arbitrary terms in TVDs, then unsoundness results. Consider the following example using `rnd()`, a pseudo-random number generator (implemented using a reference):

```
constr S = interface type T; val v:T; val f:T→T; end;

val   M1:S = module type T=int;
                    val v=3;    val f = λx:int.  -x;    end;
val   M2:S = module type T=bool;
                    val v=true; val f = λx:bool. not x; end;

val   F = λx:int. if x % 2 = 1 then M1 else M2;

val   X = module    x=F(rnd());    y=F(rnd()); end <:
            interface x:S=F(rnd());  y:S=F(rnd()); end;
```

If we allow arbitrary terms in TVDs, this example will type check, allowing us to conclude that `X.x` = `X.y` and hence that `X.x.T` = `X.y.T`; the call `X.x.f(X.y.v)` will thus type check but produce a runtime type error half the time.

If more terms than the extended values are desired in TVDs, one sound approach is to formulate a notion of *value-like terms* whose evaluation does not produce side-effects or depend on the store. Robert Harper and Chris Stone have taken this approach in recent work [24], introducing a separate judgment which they use to determine which terms are value like.

What notation of equality is appropriate for extended values (or value-like terms)? Clearly, at an absolute minimum, we need equivalent extended values (i.e., α-convertible) to be equal. For consistency, it would be nice to also allow constructors to convert using the existing constructor equality relation, but this ability is not strictly necessary. The extended-value equality must also take into account any TVDs in the context, allowing places with TVDs to be replaced by the extended value that they are declared equal to. Because of the presence of projections in extended values, it will also be necessary to handle projection (e.g., {`x=3; y=4;`}`.x` = `3`).

I see no need for including additional equalities, except possibly to handle applications if value-like terms are allowed; in that case, the implications for the decidability of value-like term equality will have to be considered carefully. No purpose seems to be served by allowing functions that are functionally equivalent to be equal, particularly given that such functions may differ in terms of efficiency.

Because translucent sums can contain constructors, equations on translucent sum values can also imply equations on constructors (e.g., `M.X.T` = `M.Y.T` in the first TVDs example). These inferences can be handled by adding the following rules for constructor equality (assuming a factored formulation of translucent sums):

$$\frac{,\ \vdash \nu_1 = \nu_2 : {<}K{>}}{,\ \vdash \nu_1! = \nu_2! :: K} \qquad \text{(E-VAL-O)}$$

$$\frac{,\ \vdash \nu_1 = \nu_2 : {<}{=}A{::}K{>}}{,\ \vdash \nu_1! = \nu_2! :: K} \qquad \text{(E-VAL-T)}$$

$$\frac{,\ \vdash A = A' :: K}{,\ \vdash {<}A{>}! = A' :: K} \qquad \text{(E-TVD)}$$

Note that constructor extractions in constructors are now of the form $\nu!$ rather than the old $x\rho!$; this change simplifies things because the value substitution operator no longer needs to handle projections or extraction — it can just substitute the value for the term variable and let the extended-value equality handle the projections and extractions.

Under the factored approach, transparent reified constructor types (${<}{=}A{::}K{>}$) are unnecessary after TVDs have been added: we can replace occurrences of **val** `x`:${<}{=}A{::}K{>}$ by **val** `x`:${<}K{>}$`=`${<}A{>}$ in interfaces while preserving the equation $x! = A$. Accordingly, under those circumstances, we could switch from the current kernel-system (*translucent*) reified constructors, which have both transparent and opaque types, to *opaque* reified constructors, which have only opaque types. This change would simplify the system and avoid the redundancy of having two different ways to specify a transparent constructor definition.

Just as the notion of reified constructors allowed translucent sums to be factored into simpler primitives in the kernel system, a notion of *singleton types* would allow factoring translucent sums with TVDs into simpler primitives. A singleton type ($(=\nu)A$) is a type that may contain only a single sort of extended values:

$$\frac{,\ \vdash \nu : A \qquad ,\ \vdash \nu = \nu' : A}{,\ \vdash \nu : (=\nu')A} \qquad \text{(ST-INTRO)}$$

Singleton types are subtypes of the appropriate non-singleton type:

$$\frac{,\ \vdash (=\nu)A :: \Omega}{,\ \vdash (=\nu)A \leq A} \qquad \text{(SI-SUB)}$$

What equality rules singleton types should obey is not completely clear, depending in part on the choice of what they can specify their members are equal to. Likely equality rules include:

$$\frac{,\ \vdash A = A' :: \Omega \qquad ,\ \vdash \nu = \nu' : A}{,\ \vdash (=\nu)A = (=\nu')A' :: \Omega} \qquad \text{(SIE-COMP)}$$

$$\frac{,\ \vdash (=\nu)\Sigma x{:}A_1.\,A_2 :: \Omega \qquad x \notin \mathrm{FTV}(\nu)}{,\ \vdash (=\nu)\Sigma x{:}A_1.\,A_2 = \Sigma x{:}(=\nu.1)A_1.\,(=\nu.2)A_2 :: \Omega} \qquad \text{(SIE-DSUM)}$$

Note that these rules allow simplifying the self function to $[\mathrm{self}{=}\nu]A = (=\nu)A$. A rule similar to E-ABBREV allows making use of the knowledge about the identity of a singleton type's members:

$$\frac{,\ \vdash x\rho \Rightarrow (=\nu)A}{,\ \vdash x\rho = \nu :: A} \qquad \text{(SI-ABBREV)}$$

If singleton types are introduced, then transparent value declarations (**val** `x`:A `=` ν) can be replaced by opaque value declarations using singleton types (**val** `x`:`(=`ν`)`A). Opaque reified constructors can then be used to handle opaque constructor declarations (**val** `x`:$<K>$) and transparent constructor declarations (**val** `x`:$(=<A>)<K>$). Thus, by using these encodings, translucent sums with TVDs can be factored into simpler components: dependent sums, singleton types, and opaque reified constructors.

13.4 Kind Components

Another possible extension is to allow translucent sums to contain kinds. For example, we might allow translucent sum types like **interface kind** `K1`; **kind** `K2=`**type**; **constr** `T1::K1`; **constr** `T2::K2`; **val** `x:T2`; **end** (unfactored) or **interface** `K1:<<>>`; `K2:<<=`**type**`>>`; `T1:<K1!>`; `T2:<K2!>`; `x:T2!`; **end** (factored) where `<<>>` and `<<=`**type**`>>` denote *reified-kind* types.

Reified kinds are similar to reified constructors, but package up kinds instead of constructors. The transparent type `<<=`K`>>` is the type of reified kinds containing the kind K (`<<`K`>>`), while the opaque type `<<>>` is the type of all reified kinds, regardless of what kind they contain. If $x\rho$ has a reified kind type under , , then the *kind extraction* $x\rho!$ is a valid kind under , . If $x\rho$ has type `<<=`K`>>` under , then $x\rho!$ and K are equal kinds under , . The type `<<=`K`>>` is a subtype of the type `<<>>` under any , for which K is a valid kind.

Both versions of this extension require adding notions of what a valid kind is and of which kinds are equal. Both notions depend on the types of the term variables in the assignment and on the equality of types judgment. This fact is likely to greatly complicate the proof theory of the extended system.

There are at least two plausible reasons to allow kind components in translucent sums. The first is based on the concept of a module as being "a chunk of namespace". If a language allows kinds to be named for abbreviation or mnemonic purposes, then it should, according to this view of what a module is, also allow kind definitions in modules.

This decision avoids the need to have different syntax for modules and let statements (including any interactive top level).

The second reason is that allowing translucent sums to contain kinds introduces *kind polymorphism* (the ability to abstract over kinds) to the language as a derived form in much the same way that normal translucent sums allow constructor polymorphism. In particular, $\Lambda\kappa.\, M$ would stand for $\lambda x{:}{<}{<}{>}{>}.\, [x!/\kappa]M$, $x \notin \mathrm{FTV}(M)$, and $M\, K$ would stand for $M\, {<}{<}K{>}{>}$.) Kind polymorphism is derivable only at the term level; $\lambda\kappa.\, A$ is not a derivable form. Kind polymorphism may be useful in combination with *singleton kinds* (defined in Section 13.8, below) to give more expressive types to higher-order functors. I discuss this further in Section 14.4.

13.5 Recursion

The kernel system provides the needed primitives to implement recursion but does not directly provide a way to build recursive functions. A real programming language should provide at least a fixed-point operator (fix), and preferably some kind of let-rec statement.

Fixed point operators can easily be implemented in the kernel system using recursive types; see Figure 13.1 for an example implementation of fix in SML-like syntax that can be used to produce recursive functions of type $\Pi x{:}A.\, A'$. Alternate implementations of fix are also possible using references instead of recursive types.

Note that this implementation of fix will work for any dependent function type, not just arrow types. Unfortunately, however, there is no way in the kernel system to abstract over just dependent function types so multiple copies of the implementation using different types may be needed to cover all the ways fix is used in a given program. It is possible, though, to handle all the non-dependent function types using a single implementation by abstracting separately over the argument and return types:

$$fix : \forall\alpha{::}\Omega.\ \forall\beta{::}\Omega.\ ((\alpha\to\beta)\to(\alpha\to\beta))\to(\alpha\to\beta)$$

One approach to the problem of needing multiple copies of fix is to simply make fix a built-in primitive with a special typing rule that allows it to work on any dependent function type:

Definition 13.5.1 (Fix Typing Rule)

$$\frac{,\ \vdash \Pi x{:}A.\, A' :: \Omega}{,\ \vdash \mathbf{fix} : ((\Pi x{:}A.\, A')\to(\Pi x{:}A.\, A'))\to(\Pi x{:}A.\, A')} \qquad \text{(T-FIX)}$$

Another approach is to introduce *parametric interfaces* ($\lambda x{:}A.\, A' :: A\Rightarrow K$), which are functions at the constructor level that map values to constructors. Parametric interfaces

```
type target_type = Πx:A. A';

type fix_type = (target_type→target_type)→target_type;

(*
 * Set things up so that we can pass a rolled up
 * version of fix1 to fix1 as its first argument:
 *)
type fix1_iter = λα::Ω. α→fix_type;

val  fix1_roll   = roll   <fix_iter>;
val  fix1_unroll = unroll <fix_iter>;

(*
 * fix1 is like fix except that it takes a rolled
 * up copy of itself as an extra argument:
 *)
fun  fix1 (rolled_fix1:rec fix1_iter) (f:target_type) =
     f (λx:A.(fix1_unroll rolled_fix1) rolled_fix1 f x);

val  fix = fix1 (fix1_roll fix1);
```

Figure 13.1: A sample fixed-point operator implementation

allow abstracting over dependent function types by first abstracting away the argument type (a normal type), and then second, abstracting away the result type modulo the argument (a parametric interface):

$$\mathbf{fix} : \forall\alpha{::}\Omega.\ \forall\beta{::}\alpha{\Rightarrow}\Omega.\ ((\Pi x{:}\alpha.\,\beta\,x)\to\Pi x{:}\alpha.\,\beta\,x)\to\Pi x{:}\alpha.\,\beta\,x$$

Parametric interfaces (often called parametric signatures when they are available only at the level of modules) are also likely to be useful for describing the types of modules; for example, they allow abstracting over a module that several components of another module depend on. This ability is most likely to be useful if value sharing is allowed. (If a module depends only on the type components of another module, then we could alternatively abstract over each type component in turn, giving the same effect at the cost of more effort.)

Adding parametric interfaces would definitely complicate the kernel system (e.g., kinds would then depend on constructors and hence term variables). How best to go about adding parametric interfaces and what the full consequences of adding them would be are open questions.

Because the kernel system provides recursive types and existentials (a degenerate form of translucent sums), it might seem natural to try and encode objects from object-oriented programming languages into the kernel system. Unfortunately, this idea does not work in practice because the kernel-system subtyping rules do not take all the properties of recursive types into account. For example, the following desirable inequality does not hold in the kernel system:

$$\mathbf{rec}\,\lambda\alpha{::}\Omega.\,\{x{:}\texttt{int};\ y{:}\texttt{int};\ \mathit{self}{:}\alpha;\} \le \mathbf{rec}\,\lambda\alpha{::}\Omega.\,\{x{:}\texttt{int};\ \mathit{self}{:}\alpha;\}$$

It is likely that additional subtyping rules could be added to remedy this problem but the effect those rules would have on the semi-decidability of subtyping and type checking is unclear.

Given a fixed point operator, let-rec statements can be elaborated into uses of fix and non-recursive let statements. Some programming languages (e.g., CLU) allow modules to be defined recursively in terms of each other. This feature is difficult to provide in a translucent sums framework for several reasons.

One problem is that recursive module definitions allow defining types recursively. This ability is different from the existing kernel-system recursive-type mechanism because it provides a true equality rather than just an isomorphism. For example, if we could define **type** `T=T`$\to$`T`, then `T` = `T`$\to$`T` would be derivable under suitable assignments; however, the similar equation $\mathbf{rec}\,\lambda\alpha{::}\Omega.\,\alpha\to\alpha = (\mathbf{rec}\,\lambda\alpha{::}\Omega.\,\alpha\to\alpha)\to(\mathbf{rec}\,\lambda\alpha{::}\Omega.\,\alpha\to\alpha)$ does not hold. Permitting types to be defined recursively is a problem because it makes the type system completely undecidable. The following recursive definition defines a fixed point operator at the constructor level:

$$\textbf{type}\ \texttt{Y} \ = \ \lambda f{::}(\Omega{\Rightarrow}\Omega){\Rightarrow}(\Omega{\Rightarrow}\Omega).\,f\,\lambda\alpha{::}\Omega.\,Y\,f\,\alpha$$

Using this combinator, any computation problem (including does Turing machine M halt?) can be encoded into a type-validity question.

If the possibility of defining types recursively is disallowed, this problem can be gotten around by staging module computation so that we first compute just the type components of each module non-recursively then compute the remaining components recursively. Unfortunately, however, there is no way in general to automatically perform this staging: because first-class translucent sums combine types and values in a single object, the computations of the type and value components can be arbitrarily interweaved.

More generally, it is possible to have n-stage module computations where the components defined in stage i depend only on type components from stages 0 to $i-1$ and on value components from stages 0 to i. This idea raises additional problems for automation because the data flow information needed to arrange this staging automatically is not available from just the module interfaces. (Requiring access to the source code of the referenced modules would break separate compilation.)

Thus, in summary, programmers can manually define modules recursively in terms of each other in the kernel system so long as no type components are defined in terms of themselves. A full automatic solution to the problem of handling module recursion is unlikely, but it seems possible that a simple automatable partial solution could be found that handles most of the uses of recursive modules.

13.6 Better Type Inference

The kernel system is explicitly typed, requiring the programmer to specify explicitly the type of function arguments, what constructors a polymorphic object should be instantiated to, when and how a term should be polymorphically abstracted over, and, for the inference system, when and how to use subsumption. It seems likely that some kind of type reconstruction algorithm could be found which would reduce the amount of information required.

One possible approach is to try and modify Frank Pfenning's type reconstruction algorithm for F_ω [48, 49] to work for the kernel system. The idea would be to require the programmer to provide all constructors involving translucent sums, dependent functions, or reified constructors. The reconstruction algorithm would then not have to infer these sorts of constructors.

13.7 Elaborations

A number of useful extensions are more naturally added as features that elaborate during an elaboration stage down to the core language. For example, a simple let statement, **let** `x:`A `=` M_1 **in** M_2 **end**, could be elaborated into the kernel-system expression $(\lambda x{:}A.\, M_2)\ M_1$. Elaboration often incorporates a degree of type inference as well. For example, we could make supplying a type for M_1 in let statements optional; since the kernel inference system possesses principle types (see Chapter 12), supplying this type is easy for the elaborator to do.

13.7.1 SML-like Elaborations

The language Standard ML (SML) [21] possesses many elaborations that would make useful extensions to the kernel system. These include complex let statements that allow sequential, parallel, recursive, and local bindings; a special syntax for function bindings (e.g., **fun** `inc(x:int) = x+1` rather than **val rec** `inc =` λx`:int. x+1`); tuples as sugar for records (translucent sums) with numbered fields (e.g., `(int, bool)` stands for `{1:int; 2:bool;}`); datatypes[2]; and pattern matching. The SML syntax for datatypes and pattern matching will need to be extended though in order to handle polymorphic constructors such as the list cons operator, which require instantiation to an explicit type before they can be used.

Using these SML-like elaborations, we can write code like the following polymorphic function which swaps the components of a homogeneous length-two tuple:

```
fun swap(<T::type>, (x:T, y:T)) = (y, x);
```

Because type variables are declared so frequently, it is helpful to introduce an abbreviation `'T` to stand for `<T::`**type**`>`:

```
fun swap('T, (x:T, y:T)) = (y, x);
```

The curried version of this function would look like

```
fun swap 'T (x:T, y:T) = (y, x);
```

Using this definition, `swap <int> (1,2)` evaluates to `(2,1)`. For an actual programming language, we would probably wish to replace the $<A>$ notation for reified constructors with some other syntax so that we can make use of the `<` and `>` symbols to denote the relational operators.

13.7.2 Translucent-Sum Syntax

Translucent sums are used in two main ways in the kernel system: they are used as record-like temporary objects that are created and passed around in the process of computation

[2]While datatypes were originally built into SML, recent reformulations of the language [24] treat them as elaborations in terms of simpler constructs: recursive types, tuples, one-ofs, and patterns.

and they are used as module-like permanent objects that contain chunks of program namespace. These uses are sufficiently different that they would benefit from different syntax. For record-like use, a light, concise syntax similar to that of records is desirable. When you are creating translucent sums every other line, {`x=3; y=4;`} is far preferable to **module val** `x=3;` **val** `y=4;` **end**.

For module-like use, on the other hand, a heavier syntax that stands out is appropriate: Modules are an important part of program structure and should be easily visible to a reader, not buried in a mess of symbols. Another important reason for using heavier syntax is that it allows incorporating SML's complex binding forms in the syntax: We can regard **module** *bindings* **end** as an elaboration for **let** *bindings* **in** {x_1=x_1; x_2=x_2; …; x_n=x_n;} **end** where x_1, x_2, …, x_n are the names bound in *bindings* in the order that they are bound.[3]

For example, **module val** `x=2;` **val** `y=x+2;` **end** would elaborate to **let val** `x=2;` **val** `y=x+2;` **in** {`x=x; y=y;`} **end**. A slight extension of the SML binding syntax will be needed here to allow for optionally different external and internal names. As an example, **module val** `x(tmp)=2;` **val** `y=x+tmp;` **end** might elaborate to **let val** `tmp=2;` **val** `y=x+tmp;` **in** {`x=tmp; y=y;`} **end**. Multiple bindings using the same external name should be disallowed. Corresponding forms of syntax are useful at the type level (e.g., {`x:int;`} vs. **interface val** `x:int;` **end**).

13.7.3 Open and Include

A useful additional form of binding is the *open* binding form (**open** M), which makes each component of a selected module available under its external name. (Pascal's with statement [26] provides a similar form of binding for records.) The elaboration of **open** M depends on M's principle type.[4] For example, if M has principle type {`x:int; y:bool;`}, then **let open** M **in** M' **end** will elaborate to **let local** `m=`M **in val** `x=m.x` **and val** `y=m.y;` **end in** M' **end**.

In combination with the previous module syntax, the open binding form allows for easy module extension. For example, **module val** `x=3;` **open** `M;` **end** creates a module identical to `M` except for an added initial `x` field equal to 3. If a similar ability is desired for interfaces, the SML **include** feature can be provided as well. (**interface val** `x:int;` **include** {`y:int; z:int;`}; **end** elaborates to **interface** `x:int; y:int; z:int;` **end**.)

[3]I am assuming in this section that the record-like syntax does not include any binding; if it provides sequential binding, then a slightly more complicated encoding using different external and internal names will have to be used here to avoid capture.

[4]The principle type is used to ensure that all of M's components are made available. If a non-principle type was used, it might be missing components because of subsumption.

13.7.4 Leroy's With Notation

An important abbreviation feature when dealing with translucent sum types is some version of Leroy's with notation [31]. Leroy's with notation is used to add information about the contents of a translucent sums' constructor components to its interface. For example, if `S` is a valid type that is equal to the semi-canonical type **interface type** `x`; **type** `y`; **end**, then `S` **with** `x=int` denotes the **interface type** `x=int`; **type** `y`; **end**. It is an error to attempt to redefine what a component is equal to (e.g., `S` **with** `x=int` **with** `x=int` is erroneous).

Choosing the right scoping rules for with expressions is somewhat tricky. We would like `S` **with** `y=x` to denote **interface type** `x`; **type** `y=x`; **end**, but what should `S` **with** `x=y` denote? Possibilities include **interface type** `x=y`; **type** `y`; **end** (here `x` is declared equal to an `x` bound outside), an error (because the inner `y` is not in scope for the declaration of `x`), **interface type** `x`; **type** `y=x`; **end** (this interfaces makes the `x` and `y` components equal), and **interface type** `y`; **type** `x=y`; **end** (this type is a subtype of `S` under some forms of extended matching; see Section 13.2 for details). Also problematic is the question of how to use with notation to set `S`'s `y` component equal to an `x` bound outside without incurring capture by `S`'s `x` component.

One solution is to alter with expressions so that they reference module components using a name supplied by the programmer. For example, `S` **with** `(it) it.y = it.x` would denote the old `S` **with** `y=x` while `S` **with** `(it) it.y=x` would denote setting the `y` component equal to an `x` bound outside of `S`. In order to keep things simple and to catch as many programmer errors as possible, I recommend declaring references to non-preceding components as errors. Thus, `S` **with** `(it) it.x = it.y` would be erroneous.

This change to the with notation solves the scoping problems at the cost of a more verbose notation. A reasonable alternative might be to provide both forms, with the original with notation as an abbreviation for the new one: `S` **with** `n=`A would stand for `S` **with** `(it) it.n=`$[\mathtt{it.x}/\mathtt{x}][\mathtt{it.y}/\mathtt{y}]A$, where `it` $\notin \mathrm{FTV}(A)$. Most of the time the shorter version could be used, but when necessary (to avoid capture) or to make things clearer, the longer version could be used.

Note that `S` **with** `z=int` is illegal, even if `z` exists outside: the with notation is only allowed to add information to its argument type, not to the surrounding context. We can extend the with notation to handle nested modules though, by allowing with to add information about subcomponents. For example, **interface type** `r`; **val** `s:S`; **end** **with** `(it) it.s.y = it.s.x->it.r` would be defined equal to **interface type** `r`; **val** `s:S` **with** `(s) s.y = s.x->r`; **end**, which in turn is equal to **interface type** `r`; **val** `s:`**interface type** `x`; **type** `y=x->r`; **end**; **end**.

I recommend against trying to implement the SML sharing-type construct because of its complicated non-local effects: **interface type** `T` **sharing type** `x=y`; **end** can set `x` and `y`, which are bound somewhere outside this type, to be equal to each other.

The clearer with construct can be used for all the same purposes that the sharing-type construct is used for in SML, abet requiring a bit more thought to use in some cases.

13.7.5 Separate Compilation

Another important extension that can be added by elaboration is separate compilation. This addition requires that the compiler automatically convert files containing code with references to other modules into closed functors using the files containing the interfaces for the referenced modules. Some kind of convention needs to be established so that the compiler can find the files containing the appropriate interfaces. The linker then must automatically generate code to apply the compiled functors generated by the compiler to each other in the correct order at runtime. There are many possible ways of setting up this sort of system; I gave an example of one way in Section 3.8.

13.8 Reformulations

Finally, I want to discuss briefly two possible reformulations of my kernel system. The first reformulation involves using *singleton kinds* as an alternative way of capturing the notion of transparent constructor declarations. Singleton kinds are due independently to Robert Harper [16] and Xavier Leroy [31]. A singleton kind is composed of a *base kind* and a constructor of that kind; in order for a constructor to belong to a singleton kind, it must belong to the singleton kind's base kind and be equal to the singleton kind's constructor. A singleton kind thus describes a single equivalence class of constructors. Singleton kinds are similar to singleton types, but involve the kind and constructor levels rather than the type and term levels.

I shall describe briefly one formulation of singleton kinds here. To simplify things, I shall restrict base kinds (B) syntactically so that they do not include singleton kinds ($=A{::}B$):

Definition 13.8.1 (Extended kind level syntax)

Base Kinds	B	$::=$	$\Omega \mid B{\Rightarrow}B'$
Kinds	K	$::=$	$\Omega \mid =A{::}B \mid K{\Rightarrow}K'$

This restriction avoids the need to handle singleton kinds of the form $=A{::}(=A'{::}K)$.

Adding singleton kinds requires allowing kinds to contain constructors; because constructors are not always valid (because their use of type and term variables may differ from what the assignment says is legal), this change means that an additional judgment for establishing kind validity ($\Gamma \vdash K$ valid) must be added:

Definition 13.8.2 (Valid Kind Rules)

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \Omega\ \mathit{valid}} \qquad \text{(KV-TYPE)}$$

$$\frac{,\ \vdash K_1\ \mathit{valid} \qquad ,\ \vdash K_2\ \mathit{valid}}{,\ \vdash K_1{\Rightarrow}K_2\ \mathit{valid}} \qquad \text{(KV-FUN)}$$

$$\frac{,\ \vdash A :: B}{,\ \vdash =A{::}B\ \mathit{valid}} \qquad \text{(KV-SINGLE)}$$

The other judgments must then be altered to ensure that all kinds in valid judgments are themselves valid. This change increases the difficulty of proving things about the system because it makes the kind level dependent on the constructor and term levels.

Because the equality relation on constructors is not syntactic identity, an equal kind judgment ($,\ \vdash K_1 = K_2$) must also be introduced:

Definition 13.8.3 (Kind Equality Rules)

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \Omega = \Omega} \qquad \text{(KE-TYPE)}$$

$$\frac{,\ \vdash K_1 = K_1' \qquad ,\ \vdash K_2 = K_2'}{,\ \vdash K_1{\Rightarrow}K_2 = K_1'{\Rightarrow}K_2'} \qquad \text{(KE-FUN)}$$

$$\frac{,\ \vdash A = A' :: B}{,\ \vdash (=A{::}B) = (=A'{::}B)} \qquad \text{(KE-SINGLE)}$$

The constructor equality judgment then needs to be modified to allow for different but equal kinds. For example, the E-LAM rule becomes

$$\frac{\begin{array}{c},\ \vdash K_1 = K_2 \\ ,\ ,\alpha{::}K_1 \vdash A_1 = A_2 :: K'\end{array}}{,\ \vdash \lambda\alpha{::}K_1.\,A_1 = \lambda\alpha{::}K_2.\,A_2 :: K_1{\Rightarrow}K'} \qquad \text{(E-LAM-K)}$$

It is also necessary to add a rule to handle checking constructor equality at singleton kinds:

$$\frac{,\ \vdash A_1 = A_2 :: B \qquad ,\ \vdash A_1 = A :: B}{,\ \vdash A_1 = A_2 :: (=A{::}B)} \qquad \text{(E-SINGLE)}$$

In order to allow subtyping to forget the information about constructor identity contained in singleton kinds, it is necessary to introduce a *subkinding* judgment ($,\ \vdash K \leq K'$) that allows singleton kinds to be converted to their base kinds:

Definition 13.8.4 (Subkinding Rules)

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \Omega \leq \Omega} \qquad \text{(KS-TYPE)}$$

$$\frac{,\ \vdash K_1' \leq K_1 \qquad ,\ \vdash K_2 \leq K_2'}{,\ \vdash K_1 \Rightarrow K_2 \leq K_1' \Rightarrow K_2'} \qquad \text{(KS-FUN)}$$

$$\frac{,\ \vdash A = A' :: B}{,\ \vdash =A{::}B \leq =A'{::}B} \qquad \text{(KS-SINGLE)}$$

$$\frac{,\ \vdash =A{::}B\ \ \mathit{valid}}{,\ \vdash =A{::}B \leq B} \qquad \text{(KS-FORGET)}$$

The subtyping and one-step subtyping judgments are then modified to allow for subkinding. For example, the following rule needs to be added:

$$\frac{,\ \vdash K_1 \leq K_2}{,\ \vdash <K_1> \lhd <K_2>} \qquad \text{(O-REIFY)}$$

Subkinding also interacts with the constructor validity judgment by means of a subsumption rule:

$$\frac{,\ \vdash A :: K' \qquad ,\ \vdash K' \leq K}{,\ \vdash A :: K} \qquad \text{(C-SUMP)}$$

The rules for introducing and using singleton kinds are then as follows:

$$\frac{,\ \vdash A = A' :: B}{,\ \vdash A :: (=A'{::}B)} \qquad \text{(K-INTRO)}$$

$$\frac{,\ \vdash A :: (=A'{::}B)}{,\ \vdash A = A' :: B} \qquad \text{(K-ABBREV)}$$

The effect of a transparent constructor declaration (`constr T=`A) can then be gotten by writing `constr T::(`$=A{::}B$`)`, where B is the base kind of A. Subsumption can be used as before to forget `T`'s identity. This ability means that we could either switch to simpler opaque reified constructors (like the case with singleton types in Section 13.3) or to a simpler version of translucent sums with only opaque constructor and value declarations.

Thus, in summary, the first possible reformulation of the kernel system is to use singleton kinds rather than transparent reified constructor types to express transparent constructor definitions. While this change would result in a more orthogonal system since the concepts of transparent constructor definitions and reified constructors would be split into separate primitives, the accompanying increase in complexity (e.g., compare the above rules to the only rules needed for transparent reified constructors: C-EXT-T, E-TRANS, E-ABBREV, and O-FORGET) seems too high to be worth paying. This tradeoff may change, however, if features like bounded polymorphism, which can be implemented using *power kinds* [9], are added to the language.

The second possible reformulation of the kernel system is to reformulate the system so that modules are second class instead of first class. This change would involve introducing separate module and interface levels; translucent sums and dependent functions would live on the module level while their "types" would live on the interface level. Because modules would no longer be terms and interfaces would no longer be types, translucent sums would need to be extended with three new kinds of declarations: opaque interface declarations, transparent interface declarations, and opaque module declarations. This extension would allow for modules like **module type** `T=int`; **val** `x=3`; **inter** `S=IO_INTERFACE`; **mod** `s=IO_MODULE` **end** and interfaces like **interface type** `O`; **type** `T=int`; **val** `x:T`; **inter** `I`; **inter** `S=IO_INTERFACE`; **mod** `s:S` **end**.

Either a non-factored version of translucent sums could be used (see Section 13.1) or opaque reified values, translucent reified constructors, and translucent reified interfaces could be introduced on the module level. These would allow packaging up a value, a constructor, or an interface as a module, possibly with information about its contents in its interface if it packages a constructor or an interface. They would thus allow opaque value, transparent and opaque constructor, and transparent and opaque interface declarations to be expressed using only opaque module declarations.

Separate record, non-dependent function, and polymorphic abstraction constructs would need to be introduced at the term level to handle the previous non-module uses of translucent sums and dependent functions. The subtyping relation on constructors would no longer be needed, but new notions of equality and subtyping for interfaces would have to be introduced. (Forgetting of information would occur only at the interface level.)

Unless additional constructs were added at the interface level (e.g., parametric interfaces or functions mapping interfaces to interfaces), these relations would be relatively simple because there is no analog to $\beta\eta$-reduction on the interface level, only an analog of γ-reduction. The subtyping relation for interfaces would still be undecidable, however, for the same reasons that the kernel system's subtyping relation is undecidable. (The proof in Sections 10.6 to 10.9 can be adapted to this case.)

An even simpler system can be obtained by disallowing opaque or transparent interface declarations (and hence γ-reductions); this change makes interface equality completely structural: equality cannot change the shape of interfaces. It thus greatly simplifies place lookup as well as many other parts of the system. It particular, it is likely that type checking decidable is decidable for this system. Xavier Leroy's manifest-types system [31] (see Section 14.3) takes this approach. It is unclear, though, how problematic the lack of transparent interface definitions is from the programmer's viewpoint.

Either way, I expect the resulting system to be far easier to prove things about in spite of the likely larger number of rules and constructs. Because there are no conditionals at the module level, types at runtime will not be able to depend on runtime values in a second-class module system; this fact means that additional typings can be assigned to such programs that would be unsound in the presence of first-class modules. One extension that permits such typings is Xavier Leroy's *applicative functors* [32]. (See further discussion in Section 14.4.) The utility of such extensions should be investigated for any second-class module system.

Chapter 14

Related Work

In this chapter I discuss work related to the work contained in this dissertation.

14.1 First-Order Module Systems

How does the module system I built relate to first-order module systems? The answer is that languages like Modula-2 [56], which have first-order module systems, basically implement the first-order fragment of a second-class version of my module system. (By the first-order fragment, I mean here the fragment that results from syntactically disallowing the use of functors and submodules.)

These languages allow translucent interfaces (interfaces that may contain both opaque and transparent type declarations), and propagate type-identity information across module boundaries similarly to my translucent approach. In particular, the only information available about a module's type component is that provided by the module's interface. Their type systems are extremely simple compared to my kernel system because they do not need to deal with issues like how to compute the interface of a functor application or how to rewrite an interface into a form that can be used.

For efficiency reasons, most of these languages only allow reference (aka pointer) types plus possibly some other types with single–machine-word representations like integers to be held abstract. This decision allows polymorphic code to treat abstract-type values uniformly since they all occupy a single machine word, while still allowing non-abstract-type values to have multiple-word representations. By contrast, most implementations of Standard ML, which does not have this restriction, are less efficient because they represent all values as single machine words. This choice allows polymorphic code to work uniformly as before on any type, but requires types that need more than one word of storage to be represented as a pointer to data in the heap. This extra level of indirection slows down Standard ML and uses more space.

The restriction on what types can be held abstract is implemented by requiring that type components declared opaquely in interfaces must contain types that may be held abstract. This restriction could be added to my kernel system by modifying the O-FORGET subtyping rule so that it requires that the type being forgotten can be held abstract:

$$\frac{, \vdash A :: \Omega \qquad , \vdash A \text{ sword}}{, \vdash \texttt{<=}A\texttt{::}\Omega\texttt{>} \lhd \texttt{<}\Omega\texttt{>}} \qquad \text{(O-FORGET')}$$

Here, $, \vdash A$ sword is a new judgment that denotes that A's values fit in a single machine word. I have limited this rule to types because none of these languages allow opaque higher-kinds. It would not be difficult, however, to extend this rule to allow holding abstract constructors that when fully applied always produce a type that fits in a single machine word.

Ada [52] handles the problem of how to implement abstraction in an efficient manner differently: Instead of restricting what types may be held abstract, it replaces opaque type declarations with *private* type declarations. A private type declaration for α (**type** α **privately is** A) declares privately that α is equal to A. This information about α's identity is not visible to the programmer: α is an abstract type that is unequal to A, preventing the programmer from taking advantage of the fact that α must be equal to A at runtime. The information is visible to the compiler, however, allowing it to generate non-polymorphic — and hence efficient — code.

The main drawback of using private rather than opaque type declarations to implement abstraction is that they increase the amount of recompilation that needs to be done when modules are changed: With private type declarations, if the programmer changes the representation type of an ADT, all the client modules that use that ADT will have to be recompiled because their code depends on the ADT's representation type. No recompilation of client modules is required with opaque type declarations because the generated code in that case is polymorphic in the ADT's representation type.

Aside from the extra recompilation incurred, using private instead of opaque type declarations works fine in a first-order module system. Attempting to build a higher-order module system without opaque type declarations, however, leads to problems; I shall explain why in Section 14.2.2 where I cover Mesa and Cedar, which have only transparent and a weak form of private type declarations.

14.2 Previous Higher-Order Module Systems

In addition to the work I have already mentioned in Chapter 2, there are several other previous (proposed) programming languages with module systems that permit non-trivial manipulations of modules.

14.2.1 Modula-3

Modula-3 [45] provides a functor-like construct called a *generic module*. A Modula-3 generic module is a module parametrized by a series of interface names. Such a module can be *instantiated* by binding its formal interface parameters to the names of actual interfaces. Instantiation is essentially equivalent to making a textual copy of the body of the generic module with the formals replaced by the actual names; each such instantiation results in new machine code, unlike functor applications in my system.

Moreover, only actual instantiations are type checked; the generic module itself cannot be type checked because no information is available for its formals. Generic modules thus act as untyped macros. This fact means that we lose the benefits of static typing and separate compilation whenever we use Modula-3's higher-order module-system features. Also, because instantiation can be done only at compile time, generic modules must be second-class values. These drawbacks make for a highly-unsatisfactory higher-order module system.

Modula-3's method of handling abstraction differs from all other higher-order module systems I am aware of. Modula-3 has two separate sorts of types, which I call *transparently-defined* types and *revealed* types. Transparently-defined types are defined using transparent type declarations and act the same as in my system. Revealed types, by contrast, are abstract by default unless information about their identity is available in the current scope.

While most higher-order module systems use a model where information about type identities flows locally from place to place, starting from the implementation and occasionally getting restricted along the way, Modula-3 uses a model where identity information for revealed types can be given anywhere using a `reveal` declaration; global restrictions ensure that the information revealed is self consistent and that the full information about a revealed type is revealed in exactly one place. By placing the full revelation for an ADT's type in its implementation module, the programmer can thus ensure that no client module has access to the full representation information. Note that because of the need for global checks, this approach cannot type check modules using only the interfaces they reference; it is thus incompatible with separate compilation.

Unlike my system, Modula-3 supports partial abstraction: The programmer can choose to reveal only that an ADT's representation type is a subtype of some type A. This feature can be used in combination with Modula-3's object-oriented features to implement private (in the sense of C++) fields in objects. It can also be used to give two interfaces to an ADT, one for normal users that hides implementation details and one for experts that reveals a more complicated and powerful interface; this is done by revealing a smaller subtype in the expert interface than in the normal one. Partial abstraction seems to be useful only when combined with objects.

The obvious way to add partial abstraction to my system would be to add partially-opaque type declarations (**type** α `<:` A). Forgetting would then allow converting from transparent to partially-opaque to even more opaque to fully-opaque type declarations. Partially-opaque type declarations in assignments would induce subtyping inequalities in the same way that transparent type declarations induce equalities in the current system. The subtyping procedure would need to be revised to handle the resulting inequalities.

14.2.2 Mesa and Cedar

The programming languages Mesa [40, 13] and its successor Cedar [30, 54, 55, 3] fall outside my opaque verses transparent classification of higher-order module systems. These languages are unusual in that they require interfaces to fully specify the identity of all type components; it is impossible in these languages to refrain from committing to what type(s) a module will contain. Types in these languages are thus really contained in interfaces (as type abbreviations) rather than in modules since two modules with type components containing different types can never have the same interface.

Accordingly, most of the interesting uses of higher-order features are inexpressible in these languages. For example, functors are much less useful when they cannot take modules containing types as inputs[1]; the B-Tree example (see Section 1.4) is completely inexpressible in these languages, for example. Also, these languages do not provide (real) data abstraction.[2]

Because modules depend only on interfaces (all the needed type information is required to be available in the interfaces), these languages support separate compilation and could, potentially, support modules as first-class values. Modules as first-class values are fairly useless, however, if modules cannot contain types, because it is not possible in that case to choose between different ADT implementations at runtime.

14.2.3 Standard ML

Experience with the programming language Standard ML (SML) [21] has provided much of the motivation for work on higher-order module systems; SML was the first widely available real programming language with a higher-order module system. SML has a (mostly) transparent module system that provides modules, submodules, and first-order functors at a separate module level. This stratification of module-like constructs from ordinary terms limits modules to being second-class values in SML.

[1]Caveat: It is possible to get around this problem somewhat by using the System Modeller (a module configuration language for Cedar)'s ability to treat any module as a Modula-3–like generic module parameterized by the names of its input and output interfaces; however, this "solution" has the same flaws as using generic modules (see Section 14.2.1).

[2]A very weak form of data abstraction — any module can violate the abstraction by simply declaring that it wishes to do so — is available via a weak form of private type declaration.

SML has the transparent approach's advantages and disadvantages as described in Section 2.2 with one exception mentioned below. Unlike in a pure transparent approach, though, datatype expressions (a core-level data-structure mechanism) and module expressions in SML are generative: Each time they are evaluated, they give rise to a new instance with a unique identity. This generativity is implemented in the SML type system by means of a complicated, non-declarative, and hard to reason about mechanism that uses stamps.

The generativity of datatype expressions is present to ensure that types generated with abstype expressions are abstract: A datatype expression creates at the same time a new type different from any other and a set of (de)-constructors for constructing and de-constructing values of that type; an abstype expression can be thought of as sugar for first declaring a datatype, second, defining some operators using the new datatype, and third, hiding the datatype's (de)-constructors. This procedure results in a type whose values can be manipulated only using the previously-defined operations.[3]

Although intended as a core-level abstraction mechanism, it is possible, though somewhat awkward, to get data abstraction at the module level by either using an abstype expression directly, or more flexibly, by using a datatype for the representation type of the ADT then using a type coercion to forget all the (de)-constructors of the datatype outside of the ADT module. SML, thus, unlike other transparent module systems, has the advantage of providing a form of module-level data abstraction.

The generativity of modules is only important for *structure-sharing specifications*, an SML feature similar to type-sharing specifications that allows requiring that two submodules of a module are identical (see Section 13.3 for discussion of how structure-sharing specifications might be added to my system); because of the generativity of modules, the test for module identity is not whether or not two modules contain the same components, but whether or not they were created from the same evaluation step. Structure-sharing specifications are not considered to be very useful and recent proposals for new versions of SML [24, 47] suggest dropping them from the language.

In addition to structure-sharing specifications, SML allows type-sharing specifications (see Section 3.9). SML allows type-sharing specifications to refer only to type names, not arbitrary type expressions. This restriction prevents SML from giving fully transparent interfaces in many cases. The lack of fully transparent interfaces, in turn, prevents SML from providing separate compilation: It is not possible in general to give an interface for a module that captures all the information the type checker can obtain from inspecting that module's implementation directly; hence there are legal SML programs that cannot be divided into module-sized pieces that type check separately.

[3]Caveat: Abstype and the given desugaring differ on their treatment of equality: abstypes never admit equality, while the desugaring's result type admits equality iff the underlying datatype does. This difference cannot be removed because SML does not provide any way to hide automatically-generated equality functions.

The New Jersey implementation of SML (SML/NJ, version 0.93) [1] extends SML in a number of ways. In particular, it adds to the SML module system the ability to have higher-order functors, although in a way that depends on SML having modules as second-class values (see Section 14.4), and a way to make selected modules' boundaries opaque, the *abstraction* binding. Note that because abstraction bindings can only make a boundary completely opaque, they cannot be used to solve the B-Tree example or similar problems. SML/NJ also adds additional software support for *incremental recompilation* [18], which partially makes up for SML/NJ's lack of separate compilation by reducing the number of unneeded recompilations required. It is still not possible to compile without implementations of all referenced modules though.

14.2.4 Experimental Designs

Burstall and Lampson's experimental language Pebble [8] is an early transparent design; unlike later transparent work, it has modules as first-class values. It avoids unsoundness only because of a lack of effects; Burstall and Lampson suggested that side effects could be added to Pebble by first extending Pebble's type system to distinguish expressions that might cause effects from those that could not, and then, second, restricting types to expressions that could not cause effects. Pebble has no phase distinction and its type checking problem is undecidable. These problems have lead later researchers to consider the transparent approach incompatible with modules as first-class values.

Mitchell, *et al.* [42] consider an extension of the SML module system with first-class modules as a means of supporting certain object-oriented programming idioms. Their paper is primarily concerned with illustrating an interesting language design rather than with the type-theoretic underpinnings of such a language, though a brief sketch is provided.

Their system is particularly interesting, because like mine, their system displays a forced loss of typing information when using modules in conditionals and other primitives. In that system, types are divided into two universes, U_1, the universe of "normal" types like int and $bool \rightarrow int$, and U_2, the universe of module types. The loss in their system is caused by the need to apply an implicit coercion from a strong sum (which belongs to U_2) to a weak sum (which belongs to U_1) because primitives operate only on terms with types in U_1. This coercion causes a total loss of typing information. My system is more flexible than this because it only loses just enough information to ensure soundness.

Russell [7] and Poly [39] both seem likely to have some relationship with my system, but a detailed comparison seems difficult in the absence of a type-theoretical analysis of these languages (see [25] for an early attempt).

14.3 Manifest Types

The most directly relevant work to mine is Xavier Leroy's work on *manifest types* [31]. This work, done independently, uses similar ideas but differs most fundamentally from mine in that his system treats modules as second-class values. This choice greatly simplifies the theoretical complexity of his system and holds out the possibility of a decidable type system if named interfaces are prohibited. (Section 13.8 discusses these matters in more detail.)

His system also differs from mine in that his system is based on Damas-Milner style polymorphism [12] and is implicitly typed, while my kernel system is based on Girard's F_ω and is explicitly typed. He has not yet provided either a proof of soundness or a provably-correct type-checking procedure[4] for his system.

The two systems also differ due to slightly different type machinery arranged in different ways. For example, Leroy independently invented an operation he calls *strengthening* (written $A/x\rho$ in my notation) which corresponds to the self function in my system. In Leroy's original formations of manifest types [31, 32], $A/x\rho$ computes essentially the same type as my [self=$x\rho$]A except for transparent types: My self function leaves transparent types unchanged ([self=$x\rho$]<=A::K> = <=A::K>), while Leroy's original strengthening operation replaces the information already in the transparent type with information about the name of the type (<=A::K>/$x\rho$ = <=$x\rho$!::K>). In more recent formations of manifest types [33, 34], Leroy has switched to a version of strengthening that treats transparent types in the same way as my self function.

Another technical difference is that Leroy combines external and internal names for modules into a single identifier (written x_i where x is the external name and i the internal one). Leroy's system requires the user to supply only the external name for an identifier, generating the internal name automatically using stamps: Each time an external name is bound, it is assigned a fresh internal identifier name; external-name references are assigned the internal name from the innermost binding of that external name in scope. This design decision is problematic, however, because it makes it impossible for the user to refer to outer components shadowed by an inner component with the same external name (e.g., the example in Section 13.1).

14.4 Modules as Second-Class Values

Since the initial publication of my work on translucent sums [20] and Leroy's work on manifest types [31], followup work has begun to appear in the literature. A sizeable

[4]He did give a type checking procedure and a "proof" of its correctness in his paper [31], but both were later discovered to be flawed.

```
I = interface type T; end;

Apply =
  functor (F:FUNCTOR(x:I):I, A:I)
    F(A);

Int  = module type T=int;  end;
Bool = module type T=bool; end;

Id       = functor (x:I) x;
ConstInt = functor (x:I) Int;
```

Figure 14.1: Leroy's paradigmatic higher-order functor example

portion of this new work concerns the expressiveness of such systems (*translucent module systems*) when modules are second-class values.

Leroy showed in a recent paper [34] that translucent module systems capable (possibly via extension) of supporting modules as first-class values can express most (unextended) SML programs[5] so long as SML type ascriptions ($M : A$) are translated to coercion functors in the obvious way. (The translation is necessary because SML type ascriptions can drop or reorder components but not forget type information.)

This result, however, does not extend to SML/NJ because the proof breaks down in the presence of higher-order functors. To see where the problem arises, consider Figure 14.1, an example due to Leroy [32]. Here we have defined an interface `I` for modules that contain a type `T` and then defined a higher-order functor `Apply` that takes a functor (`F`) mapping `I`'s to `I`'s and an `I` (`A`), and returns the result of applying `F` to `A`. The rest of the example is composed of a number of sample modules and functors for use in testing `Apply`.

In SML/NJ (assuming the appropriate syntactical translation) and other transparent module systems with higher-order functors, `Apply` behaves in what MacQueen and Tofte [38] call a *fully-transparent* manner: That is, the type checker knows everything that could be learned from actually executing the functor application(s). In particular, it knows the following:

$$\begin{aligned}\texttt{Apply(Id, Bool).T} &= \texttt{bool}\\ \texttt{Apply(ConstInt, Bool).T} &= \texttt{int}\end{aligned}$$

[5]Leroy's proof does not cover SML code that uses structure sharing or partially-applied higher-order constructors.

SML/NJ is able to infer these type identities because it type checks functor applications by essentially executing them while ignoring any value components. Such execution must terminate because there are no conditionals or looping mechanisms at the module level in SML/NJ. By actually executing functor-application expressions, SML/NJ can figure out the exact identity of the resulting type components. Note, however, that SML/NJ's ability to do this type-identity computation depends crucially on its knowledge of the exact definition of the functors involved; because this information cannot be encoded in their types in SML/NJ, many uses of `Apply` will not type check in SML/NJ if separately compiled.

In my system and Leroy's original version of manifest types, by contrast, `Apply(F,A).T` is always an abstract type, regardless of what `F` and `A` are: `Apply` gets assigned the type

FUNCTOR `(F:`**FUNCTOR** `(x:I):I, A:I):I`

because `F(A)` is not an extended value. Functors in these systems are thus not fully transparent in the sense of [38].

This behavior should not be too surprising, however, for the case where modules are first-class values: There is in general no reasonable way to completely evaluate functor applications at type-checking time when they may involve conditionals, recursion, and the computation of arbitrary values. Even approximations of fully-transparent behavior, such as Leroy's *applicative functors* (described later in this section), that assume that types do not depend on runtime values are unsound in the presence of first-class modules. It may be possible to incorporate such approximations in a system with first-class modules by distinguishing (via the type system) those functors that do not use modules in a first-class manner; unsoundness would then be avoided by using the approximation only on expressions involving those functors.

It is possible, however, to specialize `Apply` so that it can handle some of the `Apply` test cases correctly at the cost of excluding others. For example, we could define the following specialized versions of `Apply`:

```
ApplyId =
  functor (F:FUNCTOR(x:I) :I with T=x.T, A:I)
    F(A);

ApplyConstInt =
  functor (F:FUNCTOR(x:I) :I with T=int, A:I)
    F(A);
```

Then, under the two translucent systems, we would have that

$$\begin{array}{lcl} \texttt{ApplyId(Id,Int).T} & = & \texttt{int} \\ \texttt{ApplyId(Id,Bool).T} & = & \texttt{bool} \\ \texttt{ApplyConstInt(ConstInt,Int).T} & = & \texttt{int} \\ \texttt{ApplyConstInt(ConstInt,Bool).T} & = & \texttt{int} \end{array}$$

However, `ApplyId(ConstInt,Int)` and `ApplyConstInt(Id,Int)` fail to type check.

Xavier Leroy has suggested that it may be possible to some degree to use extra parameterization by possibly-singleton kinds (see Sections 13.4 and 13.8) to produce more flexible versions of `Apply` [35]. For example, the following version can act either like `Apply` if it is given the kind **type** as its first argument or like `ApplyConstInt` if it is given the singleton kind =`int`::**type** as its first argument:

```
FlexApply =
  functor (K:KIND, F:FUNCTOR(x:I) :I with T::K, A:I)
    F(A);
```

Here, `I` **with** `T::K` is intended to denote **interface constr** `T::K;` **end** by analogy with the usual with notation. `FlexApply` cannot be made to act like `ApplyId` though (`x.T` is not in scope outside of `FlexApply`). It is possible that `FlexApply` could be made still more flexible by introducing kinds that depend on term variables but this approach seems to be far more trouble than it is worth.

If modules are restricted to second-class values, the translucent approach can be extended to handle the `Apply` example in the same way that SML/NJ does. Leroy's work on applicative functors [32] demonstrates one way to accomplish this. Under this approach, functor applications are assumed to never generate new constructors: Any result constructor (sub)-component of a functor is assumed to be a (deterministic) function of the constructors contained in that functor's arguments. Thus, if `A` = `B` and `F(A).T` is valid then `F(A).T` must equal `F(B).T`. This assumption is true in SML/NJ because SML/NJ has no way to choose a module (and hence its constructors) at runtime.

Leroy noticed that under this assumption he could safely allow the following set of terms in constructors:

$$\text{Extended Names} \quad \xi \quad ::= \quad x \mid \xi.y \mid \xi_1(\xi)_2$$

Note that because of the assumption and the fact that we are using call-by-value, the expression $\xi.\alpha$ must denote a constant constructor during each of its dynamic scopes.

Using this extension to the set of terms legal in constructors, we can give `Apply` the following type:

```
FUNCTOR (F:FUNCTOR (x:I):I, A:I):I with T=F(A).T
```

(Leroy's EVALUE-like rules give `F(A)` the type `I` **with** `T=F(A).T` in this system.) To handle type checking `Apply(Id,Bool)` then, for example, we would first specialize further `Apply`'s type as follows:

```
FUNCTOR (F:FUNCTOR (x:I):I, A:I):I with T=F(A).T

FUNCTOR (F:FUNCTOR (x:I):I with T=x.T, A:I)
  :I with T=F(A).T

FUNCTOR (F:FUNCTOR (x:I):I with T=x.T, A:I)
  :I with T=A.T
```

The final specialized type is same type as the original translucent systems gave to `ApplyId`. Accordingly, it can be used to show that

$$\begin{aligned} \texttt{Apply(Id,Int).T} &= \texttt{int} \\ \texttt{Apply(Id,Bool).T} &= \texttt{bool} \end{aligned}$$

If we wished to type check `Apply(ConstInt,Bool).T` instead, we would have instead specialized `Apply` to the type of `ApplyConstInt`:

```
FUNCTOR (F:FUNCTOR (x:I):I with T=int, A:I)
  :I with T=int
```

This method is in fact general and can be used to handle all uses of `Apply` possible in SML/NJ.

Leroy's applicative-functor system is thus more fully transparent than either of my and Leroy's original translucent module systems. Because of the applicative-functor system's underlying assumptions, functors in that system cannot produce new abstract types each time they are applied to the same argument. However, this slight loss of abstractive power over the original systems is unlikely to be problematic in practice.

Leroy's applicative-functor system is not fully transparent, however. Consider the following more complicated functor:

```
ApplyList =
  functor (F:FUNCTOR(x:I):I, A:I)
    F(module type T=list(A.T); end);
```

Because `ApplyList`'s functor body is not an extended name, `ApplyList` can only be given the opaque type

```
FUNCTOR (F:FUNCTOR (x:I):I, A:I) :I
```

Accordingly, `ApplyList(Id,Int).T` is abstract in this system rather than equal to `list(int)` as required by full transparency.

In principle it should be possible to build a system with a rich enough type system so that both separate compilation and full transparency can be achieved at the same time. Because separate compilation requires that all information needed for type checking the uses of a functor be expressible in that functor's interface, this goal will require functor interfaces to (optionally) contain an idealized copy of the code for the functor whose behavior they specify. I expect such a system to be highly complicated and hard to reason about. MacQueen and Tofte [38] have begun to explore how a system with full transparency might be built.

There thus appears to be a tradeoff in higher-order module systems with second-class modules between system simplicity and the degree of full transparency provided. Unfortunately, there is as yet no consensus on the value of full transparency to programmers because of insufficient experience with higher-order functors.

14.5 Other Follow-On Work

Feeling that the type theory of translucent module systems is unnecessarily complicated, Mark Jones has recently proposed an alternative approach to handling higher-order module systems [27]. He observes that programs written in a SML-like module system without abstraction or generativity can be translated into a system with no modules or functors but with (ordinary) records, functions, and universal polymorphism ($\forall\alpha.A$). The translation works by lifting type definitions within a module to the top level, in a manner similar to λ-lifting (see his paper for details). He believes that abstraction can be provided separately, possibly via (opaque) existentials. He thus suggests that we can do away with modules containing type components entirely, and just program directly in the translation's target system.

I do not consider his proposed solution satisfactory for three reasons. First, his transform does not respect extended-interface matching: Dropping a type component from a module does not produce a module whose type is a supertype of the original module's type. Second, his suggested style of programming forces the programmer to place the type definitions and procedure definitions of a single "module" far apart, violating the modularity percept that related definitions should be bundled together. Third, and most crucially, he is mistaken in his belief that abstraction can be separated from where types are defined: Abstraction via existentials requires that abstract types be defined in the same place as the operations that implement them while his transformation requires separating type definitions from the code that uses them; thus, his transformation and abstraction via existentials are incompatible.

My work has already begun to be used in the design of new programming languages.

In [46], John Ophel investigates how a small language with first-class modules based on translucent sums might be defined and what the consequences of such a language might be. In [24], Robert Harper and Chris Stone give a type-theoretic semantics for a new version of SML, SML 1996, using an typed intermediate language with translucent sums.

While SML 1996 is essentially a slightly better tuned and simplified version of the original SML, work is underway to build a successor language to SML, currently called ML2000, that will be substantially more powerful than the current SML implementations [17]. ML2000's module system is based on my work on translucent sums. It treats modules as second-class values in order to keep the type system simple; I feel that this was a wise choice, given that ML2000's type system also has to deal with object oriented features, a known source of complexity [47].

Chapter 15

Conclusions

In this, the final chapter, I sum up the contributions of my dissertation and discuss possible future work.

15.1 Contributions

This dissertation makes a number of contributions. I divide them up here according to the area they contribute in.

15.1.1 Better Higher-Order Module Systems

I began this dissertation work with the following thesis:

> By basing a module system on a new type-theoretic concept, namely a new kind of weak sum that can contain both transparent and opaque type definitions called a translucent sum, it is possible to obtain a higher-order module system with the advantages of both the opaque and the transparent approaches, but none of their disadvantages.

I believe I have by now firmly established this thesis. In chapter 3, I showed how my approach using translucent sums provides the advantages of the opaque approach, namely

- (Open) data abstraction is provided at the module level.
- Separate compilation is supported.
- Modules are first-class values.

as well as the advantages of the transparent approach, namely

- Modules can "contain" other types.
- Modules may export type abbreviations.

Moreover, my approach provides the following additional advantages not available in either of the two previous approaches:

- All of the higher-order idioms work.
- Transparency and opacity can be mixed, even in the same module.

My approach also offers the following advantages:

- User-defined higher-order constructors are available.
- Functors are first-class.
- Higher-order functors are provided.
- Type-sharing specifications can be encoded.
- Linking and separate compilation can be described in the resulting programming language.
- The resulting system has a simple and uniform type theory.

My approach thus represents a major step forward in the design of higher-order module systems over previous approaches, providing the first truly satisfactory higher-order module system. This is the single biggest contribution of my dissertation.

15.1.2 Feasibility of My Approach

In order to show that these advantages are actually realizable in practice, I need to show that my approach is feasible. I have done this by creating a complete working system using my approach — the kernel system — and proving that it has the necessary properties. In particular, I have shown the following:

- How to arrange the system and its associated proofs so they are tractable
- The system's soundness even in the presence of side effects
- How to do type checking effectively by giving:
  - a recursive algorithm for constructor and assignment validity
  - a recursive algorithm for constructor equality

  - a semi-decision procedure for subtyping
  - a proof that type checking is decidable, given an oracle for subtyping
  - a semi-decision procedure for type checking
  - an argument that the semi-decidability of type checking is not likely to be problem in practice

All of these represent contributions of my dissertation; the first item above is particularly important because of the complexity of systems providing translucent sums. I expect that this dissertation, especially the section on recommended improvements (Section 9.7), will prove invaluable to future designers of systems with translucent sums. In additional to being useful as a starting point for building extended systems, my kernel system is likely to be useful as a reduction target because of these proven properties; for example, I expect that the soundness of Xavier Leroy's system [31] can be established in this way far more easily than if it was proved directly.

The negative results about soundness under certain extensions, the decidability of subtyping (and hence type checking), and the existence of principle types if implicit subsumption is permitted are also contributions of my dissertation. Similarly to the case with the full kernel system, I expect that the simple system (see Section 10.6) will be useful in proving subtyping in other systems with translucent sums undecidable.

15.1.3 Technical Contributions

My dissertation also contains a number of more technical contributions. The most obvious of these are a large number of properties about the kernel system's type system that are useful in reasoning about the kernel system. For example, I have shown that in the kernel system all of the following operations preserve judgment validity:

- weakening (adding extra declarations to assignments)
- strengthening (removing unreferenced declarations from assignments)
- replacing a type in an assignment by a subtype
- subject reduction on constructors

Several of the techniques I used to make the proofs more tractable are also noteworthy:

- Specializing the well-typed term judgment to places so that the constructor validity and equality judgments do not depend on term validity or subtyping judgments
- Using the self function to normalize typing derivations

- Defining subtyping as a series of equality and one-step subtyping steps
- Factoring translucent sums into dependent sums plus reified constructors
- Introducing a separate tagged system to remove the dependency of rewriting (and hence equality) on assignments

The first three of these techniques are tied fairly closely to systems involving translucent sums. However, the fourth technique, factoring, could be used to simplify any system with modules or sums that allow fields to contain either terms or types, and the fifth technique, the use of a tagged system, could be applied usefully in any system where the rewriting of constructors depends on the assignment.

15.2 Future work

One obvious piece of future work would be to construct a prototype based on the kernel system, but with a more programmer-friendly interface. Ideally, some form of type inference would be devised for the prototype so as to reduce the amount of type information that would need to be given by the programmer. The prototype would be especially useful in evaluating the value of having modules as first-class values; exploration with the prototype might well lead to the discovery of useful new higher-order–module idioms.

The two extensions to the kernel system that I think are the most interesting to pursue are adding reordering via an elaboration stage (this will require determining the proper reordering rule(s) for subtyping), and adding transparent value declarations to permit inlining. I also think that it is worth investigating the relationship between first-class modules and objects in more detail.

Appendix A

Collected Kernel System Rules

A.1 The Kind and Constructor Level

Definition A.1.1 (Syntax for the constructor and kind levels)

Kinds	K	$::=$	$\Omega \mid K \Rightarrow K'$
Constructors	A	$::=$	$\alpha \mid \Pi x{:}A.\, A' \mid \Sigma x{:}A.\, A' \mid \lambda\alpha{::}K.\, A \mid A\, A' \mid$
			$<\!K\!> \mid <\!=\!A{::}K\!> \mid x\rho! \mid \mathbf{rec} \mid \mathbf{ref}$
Paths	ρ	$::=$	$\epsilon \mid \rho.1 \mid \rho.2$
Assignments	,	$::=$	$\bullet \mid$, , D
Declarations	D	$::=$	$\alpha{::}K \mid x{:}A$

Definition A.1.2 (Free constructor variables)

$$\begin{array}{rcl}
FCV(\alpha) & = & \{\alpha\} \\
FCV(\lambda\alpha{::}K.\,A) & = & FCV(A) \Leftrightarrow \{\alpha\} \\
FCV(\Pi x{:}A_1.\,A_2) & = & FCV(A_1) \cup FCV(A_2) \\
FCV(\Sigma x{:}A_1.\,A_2) & = & FCV(A_1) \cup FCV(A_2) \\
FCV(A_1\,A_2) & = & FCV(A_1) \cup FCV(A_2) \\
FCV(<\!=\!A{::}K\!>) & = & FCV(A) \\
FCV(<\!K\!>) & = & \emptyset \\
FCV(x\rho!) & = & \emptyset \\
FCV(\mathbf{rec}) & = & \emptyset \\
FCV(\mathbf{ref}) & = & \emptyset \\
 & & \\
FCV(\alpha{::}K) & = & \emptyset \\
FCV(x{:}A) & = & FCV(A) \\
 & & \\
FCV(\bullet) & = & \emptyset \\
FCV(,\,, D) & = & FCV(,\,) \cup FCV(D)
\end{array}$$

Definition A.1.3 (Free term variables)

$$\begin{array}{rcl}
FTV(x\rho!) & = & \{x\} \\
FTV(\Pi x{:}A_1.\,A_2) & = & FTV(A_1) \cup (FTV(A_2) \Leftrightarrow \{x\}) \\
FTV(\Sigma x{:}A_1.\,A_2) & = & FTV(A_1) \cup (FTV(A_2) \Leftrightarrow \{x\}) \\
FTV(\lambda\alpha{::}K.\,A) & = & FTV(A) \\
FTV(A_1\,A_2) & = & FTV(A_1) \cup FTV(A_2) \\
FTV(<\!=\!A{::}K\!>) & = & FTV(A) \\
FTV(\alpha) & = & \emptyset \\
FTV(<\!K\!>) & = & \emptyset \\
FTV(\mathbf{rec}) & = & \emptyset \\
FTV(\mathbf{ref}) & = & \emptyset \\
 & & \\
FTV(\alpha{::}K) & = & \emptyset \\
FTV(x{:}A) & = & FTV(A) \\
 & & \\
FTV(\bullet) & = & \emptyset \\
FTV(,\,, D) & = & FTV(,\,) \cup FTV(D)
\end{array}$$

Definition A.1.4 (Constructor substitution)

$$\begin{array}{rcll}
[A/\alpha]\alpha & = & A & \\
[A/\alpha]\alpha' & = & \alpha' & (\alpha \neq \alpha') \\
[A/\alpha]\lambda\alpha'{::}K.\,A' & = & \lambda\alpha'{::}K.\,[A/\alpha]A' & (\alpha' \neq \alpha,\ \alpha' \notin FCV(A)) \\
[A/\alpha]\Pi x{:}A_1.\,A_2 & = & \Pi x{:}[A/\alpha]A_1.\,[A/\alpha]A_2 & (x \notin FTV(A)) \\
[A/\alpha]\Sigma x{:}A_1.\,A_2 & = & \Sigma x{:}[A/\alpha]A_1.\,[A/\alpha]A_2 & (x \notin FTV(A)) \\
[A/\alpha](A_1\,A_2) & = & [A/\alpha]A_1\,[A/\alpha]A_2 & \\
[A/\alpha]{<}{=}A'{::}K{>} & = & {<}{=}[A/\alpha]A'{::}K{>} & \\
[A/\alpha]{<}K{>} & = & {<}K{>} & \\
[A/\alpha]x\rho! & = & x\rho! & \\
[A/\alpha]\mathbf{rec} & = & \mathbf{rec} & \\
[A/\alpha]\mathbf{ref} & = & \mathbf{ref} & \\
 & & & \\
[A/\alpha](\alpha'{::}K) & = & \alpha'{::}K & \\
[A/\alpha](x{:}A') & = & x{:}[A/\alpha]A' & \\
 & & & \\
[A/\alpha]\bullet & = & \bullet & \\
[A/\alpha](,\,,D) & = & ([A/\alpha],\,),[A/\alpha]D &
\end{array}$$

Definition A.1.5 (Place substitution)

$$\begin{array}{rcll}
[x\rho/x']x'\rho' & = & x\rho\rho' & \\
[x\rho/x']x''\rho'' & = & x''\rho'' & (x' \neq x'') \\
 & & & \\
[x\rho/x'](x''\rho''!) & = & ([x\rho/x']x''\rho'')! & \\
[x\rho/x']\Pi x''{:}A_1.\,A_2 & = & \Pi x''{:}[x\rho/x']A_1.\,[x\rho/x']A_2 & (x'' \neq x,\ x'' \neq x') \\
[x\rho/x']\Sigma x''{:}A_1.\,A_2 & = & \Sigma x''{:}[x\rho/x']A_1.\,[x\rho/x']A_2 & (x'' \neq x,\ x'' \neq x') \\
[x\rho/x']\lambda\alpha{::}K.\,A & = & \lambda\alpha{::}K.\,[x\rho/x']A & \\
[x\rho/x'](A_1\,A_2) & = & [x\rho/x']A_1\,[x\rho/x']A_2 & \\
[x\rho/x']{<}{=}A'{::}K{>} & = & {<}{=}[x\rho/x']A'{::}K{>} & \\
[x\rho/x']\alpha & = & \alpha & \\
[x\rho/x']{<}K{>} & = & {<}K{>} & \\
[x\rho/x']\mathbf{rec} & = & \mathbf{rec} & \\
[x\rho/x']\mathbf{ref} & = & \mathbf{ref} & \\
 & & & \\
[x\rho/x'](\alpha{::}K) & = & \alpha{::}K & \\
[x\rho/x'](x''{:}A) & = & x''{:}[x\rho/x']A & \\
 & & & \\
[x\rho/x']\bullet & = & \bullet & \\
[x\rho/x'](,\,,D) & = & ([x\rho/x'],\,),[x\rho/x']D &
\end{array}$$

Definition A.1.6 (Selection)

$$\begin{array}{rcl} \mathcal{S}(A, x\rho, \epsilon) & = & A \\ \\ \mathcal{S}(\Sigma x'{:}A_1.\,A_2, x\rho, .1\rho') & = & \mathcal{S}(A_1, x\rho.1, \rho') \\ \mathcal{S}(\Sigma x'{:}A_1.\,A_2, x\rho, .2\rho') & = & \mathcal{S}([x\rho.1/x']A_2, x\rho.2, \rho') \end{array}$$

Definition A.1.7 (Assignment regarded as a partial function)

$$\begin{array}{rcll} \mathit{dom}(\bullet) & = & \emptyset & \\ \mathit{dom}(,\ , x{:}A) & = & \mathit{dom}(,\ ) \cup \{x\} & \\ \mathit{dom}(,\ , \alpha{::}K) & = & \mathit{dom}(,\ ) \cup \{\alpha\} & \\ \\ (,\ _1; x{:}A; ,\ _2)(x) & = & A & \qquad (x \notin \mathit{dom}(,\ _1)) \end{array}$$

Definition A.1.8 (Arrow types)
The arrow type $A{\to}A'$ is defined to be equal to $\Pi x{:}A.\,A'$ where $x \notin FTV(A')$. The arrow operator $(\to)$ is defined to have lower precedence than application (A_1A_2).

Definition A.1.9 (Pair types)
The pair type (A, A') is defined to be equal to $\Sigma x{:}A.\,A'$ where $x \notin FTV(A')$.

Definition A.1.10 (Polymorphic types)
The polymorphic type $\forall\alpha{::}K.A$ is defined to be equal to $\Pi x{:}{<}K{>}.\,[x!/\alpha]A$ where $x \notin FTV(A)$.

Definition A.1.11 (Judgments)

$\vdash ,\ \ \mathit{valid}$	*valid assignment*
$,\ \vdash A :: K$	*valid constructor*
$,\ \vdash A = A' :: K$	*equal constructors*
$,\ \vdash A \leq A'$	*subtype relation*
$,\ \vdash x\rho \Rightarrow A$	*place lookup*
$,\ \vdash A \lhd A'$	*one-step subtype relation*

Definition A.1.12 (Assignment Formation Rules)

$$\vdash \bullet\ \mathit{valid} \qquad \text{(EMPTY)}$$

$$\frac{\vdash ,\ \ \mathit{valid} \qquad \alpha \notin \mathit{dom}(,\ )}{\vdash ,\ , \alpha{::}K\ \mathit{valid}} \qquad \text{(DECL-C)}$$

$$\frac{,\ \vdash A :: \Omega \qquad x \notin dom(,\ )}{\vdash ,\ , x{:}A \ \mathit{valid}} \tag{DECL-T}$$

Definition A.1.13 (Constructor Formation Rules)

$$\frac{\vdash ,\ \ \mathit{valid} \qquad \alpha{::}K \in ,}{,\ \vdash \alpha :: K} \tag{C-VAR}$$

$$\frac{,\ , x{:}A \vdash A' :: \Omega}{,\ \vdash \Pi x{:}A.\, A' :: \Omega} \tag{C-DFUN}$$

$$\frac{,\ , x{:}A \vdash A' :: \Omega}{,\ \vdash \Sigma x{:}A.\, A' :: \Omega} \tag{C-DSUM}$$

$$\frac{,\ , \alpha{::}K \vdash A :: K'}{,\ \vdash \lambda\alpha{::}K.\, A :: K{\Rightarrow}K'} \tag{C-LAM}$$

$$\frac{,\ \vdash A_1 :: K_2{\Rightarrow}K \qquad ,\ \vdash A_2 :: K_2}{,\ \vdash A_1\, A_2 :: K} \tag{C-APP}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash {<}K{>} :: \Omega} \tag{C-OPAQ}$$

$$\frac{,\ \vdash A :: K}{,\ \vdash {<}{=}A{::}K{>} :: \Omega} \tag{C-TRANS}$$

$$\frac{,\ \vdash x\rho \Rightarrow {<}K{>}}{,\ \vdash x\rho! :: K} \tag{C-EXT-O}$$

$$\frac{,\ \vdash x\rho \Rightarrow {<}{=}A{::}K{>}}{,\ \vdash x\rho! :: K} \tag{C-EXT-T}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{rec} :: (\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega} \tag{C-REC}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{ref} :: \Omega{\Rightarrow}\Omega} \tag{C-REF}$$

Definition A.1.14 (Constructor Equality Rules)

$$\frac{,\ \vdash A :: K}{,\ \vdash A = A :: K} \qquad \text{(E-REFL)}$$

$$\frac{,\ \vdash A' = A :: K}{,\ \vdash A = A' :: K} \qquad \text{(E-SYM)}$$

$$\frac{,\ \vdash A = A' :: K \qquad ,\ \vdash A' = A'' :: K}{,\ \vdash A = A'' :: K} \qquad \text{(E-TRAN)}$$

$$\frac{\begin{array}{c},\ \vdash A_1 = A_1' :: \Omega \\ ,\ , x{:}A_1 \vdash A_2 = A_2' :: \Omega\end{array}}{,\ \vdash \Pi x{:}A_1.\, A_2 = \Pi x{:}A_1'.\, A_2' :: \Omega} \qquad \text{(E-DFUN)}$$

$$\frac{\begin{array}{c},\ \vdash A_1 = A_1' :: \Omega \\ ,\ , x{:}A_1 \vdash A_2 = A_2' :: \Omega\end{array}}{,\ \vdash \Sigma x{:}A_1.\, A_2 = \Sigma x{:}A_1'.\, A_2' :: \Omega} \qquad \text{(E-DSUM)}$$

$$\frac{,\ , \alpha{::}K \vdash A = A' :: K'}{,\ \vdash \lambda\alpha{::}K.\, A = \lambda\alpha{::}K.\, A' :: K{\Rightarrow}K'} \qquad \text{(E-LAM)}$$

$$\frac{\begin{array}{c},\ \vdash A_2 = A_2' :: K \\ ,\ \vdash A_1 = A_1' :: K{\Rightarrow}K'\end{array}}{,\ \vdash A_1\, A_2 = A_1'\, A_2' :: K'} \qquad \text{(E-APP)}$$

$$\frac{,\ \vdash A = A' :: K}{,\ \vdash {<}{=}A{::}K{>} = {<}{=}A'{::}K{>} :: \Omega} \qquad \text{(E-TRANS)}$$

$$\frac{,\ , \alpha{::}K \vdash A :: K' \qquad ,\ \vdash A' :: K}{,\ \vdash (\lambda\alpha{::}K.\, A)\, A' = [A'/\alpha]A :: K'} \qquad \text{(E-BETA)}$$

$$\frac{,\ \vdash A :: K{\Rightarrow}K' \qquad \alpha \notin FCV(A)}{,\ \vdash \lambda\alpha{::}K.\, A\, \alpha = A :: K{\Rightarrow}K'} \qquad \text{(E-ETA)}$$

$$\frac{,\ \vdash x\rho! :: K \qquad ,\ \vdash x\rho \Rightarrow A \qquad ,\ \vdash A = {<}{=}A'{::}K'{>} :: \Omega}{,\ \vdash x\rho! = A' :: K} \qquad \text{(E-ABBREV)}$$

Definition A.1.15 (Place Lookup Rules)

$$\frac{\vdash ,\ \mathit{valid} \qquad x{:}A \in ,}{,\ \vdash x \Rightarrow A} \qquad \text{(P-INIT)}$$

$$\frac{,\ \vdash x\rho \Rightarrow A \qquad ,\ \vdash A = A' :: \Omega \qquad A'' = \mathcal{S}(A', x\rho, \rho')}{,\ \vdash x\rho\rho' \Rightarrow A''} \qquad \text{(P-MOVE)}$$

Definition A.1.16 (Subtyping Rules)

$$\frac{,\ \vdash A = A' :: \Omega}{,\ \vdash A \leq A'} \qquad \text{(S-EQ)}$$

$$\frac{,\ \vdash A \lhd A'}{,\ \vdash A \leq A'} \qquad \text{(S-ONE)}$$

$$\frac{,\ \vdash A \leq A' \qquad ,\ \vdash A' \leq A''}{,\ \vdash A \leq A''} \qquad \text{(S-TRAN)}$$

Definition A.1.17 (One-Step Subtyping Rules)

$$\frac{,\ \vdash A :: \Omega}{,\ \vdash A \lhd A} \qquad \text{(O-REFL)}$$

$$\frac{\begin{array}{c},\ \vdash A_1' \lhd A_1 \\ ,\ ,x{:}A_1' \vdash A_2 \lhd A_2' \qquad ,\ ,x{:}A_1 \vdash A_2 :: \Omega\end{array}}{,\ \vdash \Pi x{:}A_1.\, A_2 \lhd \Pi x{:}A_1'.\, A_2'} \qquad \text{(O-DFUN)}$$

$$\frac{\begin{array}{c},\ \vdash A_1 \lhd A_1' \\ ,\ ,x{:}A_1 \vdash A_2 \lhd A_2' \qquad ,\ ,x{:}A_1' \vdash A_2' :: \Omega\end{array}}{,\ \vdash \Sigma x{:}A_1.\, A_2 \lhd \Sigma x{:}A_1'.\, A_2'} \qquad \text{(O-DSUM)}$$

$$\frac{,\ \vdash A :: K}{,\ \vdash {<}{=}A{::}K{>} \lhd {<}K{>}} \qquad \text{(O-FORGET)}$$

A.2 The Term Level

Definition A.2.1 (Syntax for the term level)

$$\begin{array}{lll} \textit{Terms} \quad M & ::= & x \mid \lambda x{:}A.\, M \mid M_1\, M_2 \mid (M_1,\, M_2) \mid M.1 \mid M.2 \mid {<}A{>} \mid \\ & & M{<:}A \mid \textbf{roll} \mid \textbf{unroll} \mid \textbf{new} \mid \textbf{get} \mid \textbf{set} \end{array}$$

Definition A.2.2 (Free constructor variables)

$$
\begin{array}{rcl}
FCV(x) & = & \emptyset \\
FCV(\mathbf{roll}) & = & \emptyset \\
FCV(\mathbf{unroll}) & = & \emptyset \\
FCV(\mathbf{new}) & = & \emptyset \\
FCV(\mathbf{get}) & = & \emptyset \\
FCV(\mathbf{set}) & = & \emptyset \\
 & & \\
FCV(M.1) & = & FCV(M) \\
FCV(M.2) & = & FCV(M) \\
FCV(<A>) & = & FCV(A) \\
 & & \\
FCV(\lambda x{:}A.\, M) & = & FCV(A) \cup FCV(M) \\
FCV(M<{:}A) & = & FCV(M) \cup FCV(A) \\
FCV(M_1\, M_2) & = & FCV(M_1) \cup FCV(M_2) \\
FCV((M_1,\, M_2)) & = & FCV(M_1) \cup FCV(M_2)
\end{array}
$$

Definition A.2.3 (Free term variables)

$$
\begin{array}{rcl}
FTV(x) & = & \{x\} \\
FTV(\lambda x{:}A.\, M) & = & FTV(A) \cup (FTV(M) \Leftrightarrow \{x\}) \\
 & & \\
FTV(\mathbf{roll}) & = & \emptyset \\
FTV(\mathbf{unroll}) & = & \emptyset \\
FTV(\mathbf{new}) & = & \emptyset \\
FTV(\mathbf{get}) & = & \emptyset \\
FTV(\mathbf{set}) & = & \emptyset \\
 & & \\
FTV(M.1) & = & FTV(M) \\
FTV(M.2) & = & FTV(M) \\
FTV(<A>) & = & FTV(A) \\
 & & \\
FTV(M_1\, M_2) & = & FTV(M_1) \cup FTV(M_2) \\
FTV((M_1,\, M_2)) & = & FTV(M_1) \cup FTV(M_2) \\
FTV(M<{:}A) & = & FTV(M) \cup FTV(A)
\end{array}
$$

Definition A.2.4 (Judgments)

$, \vdash M : A$ *well-typed term*

Definition A.2.5 (Term Formation Rules)

$$\frac{,\ \vdash ,\ (x) = A :: \Omega}{,\ \vdash x : [\mathit{self}{=}x]A} \qquad \text{(T-VAR)}$$

$$\frac{,\ , x{:}A \vdash M : A'}{,\ \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, A'} \qquad \text{(T-LAM)}$$

$$\frac{,\ \vdash M_1 : A_2 {\rightarrow} A \qquad ,\ \vdash M_2 : A_2}{,\ \vdash M_1\, M_2 : A} \qquad \text{(T-APP)}$$

$$\frac{,\ \vdash M_1 : A_1 \qquad ,\ \vdash M_2 : A_2}{,\ \vdash (M_1,\, M_2) : (A_1, A_2)} \qquad \text{(T-PAIR)}$$

$$\frac{,\ \vdash M : \Sigma x{:}A_1.\, A_2}{,\ \vdash M.1 : A_1} \qquad \text{(T-FST)}$$

$$\frac{,\ \vdash M : (A_1, A_2)}{,\ \vdash M.2 : A_2} \qquad \text{(T-SND)}$$

$$\frac{,\ \vdash A :: K}{,\ \vdash {<}A{>} : {<}{=}A{::}K{>}} \qquad \text{(T-REIFY)}$$

$$\frac{,\ \vdash M : A' \qquad ,\ \vdash A' \leq A}{,\ \vdash M : A} \qquad \text{(T-SUMP)}$$

$$\frac{,\ \vdash M : A}{,\ \vdash M{<}{:}A : A} \qquad \text{(T-COERCE)}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{new} : \forall\alpha{::}\Omega.\alpha{\rightarrow}\mathbf{ref}\,\alpha} \qquad \text{(T-NEW)}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{get} : \forall\alpha{::}\Omega.\mathbf{ref}\,\alpha{\rightarrow}\alpha} \qquad \text{(T-GET)}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{set} : \forall\alpha{::}\Omega.\mathbf{ref}\,\alpha{\rightarrow}(\alpha{\rightarrow}\alpha)} \qquad \text{(T-SET)}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{roll} : \forall\alpha{::}\Omega{\Rightarrow}\Omega.\alpha\,(\mathbf{rec}\,\alpha){\rightarrow}\mathbf{rec}\,\alpha} \qquad \text{(T-ROLL)}$$

$$\frac{\vdash ,\ \ \mathit{valid}}{,\ \vdash \mathbf{unroll} : \forall\alpha{::}\Omega{\Rightarrow}\Omega.\mathbf{rec}\,\alpha{\rightarrow}\alpha\,(\mathbf{rec}\,\alpha)} \qquad \text{(T-UNROLL)}$$

Definition A.2.6 (Self function)

$$
\begin{array}{rcll}
[\mathit{self}{=}x\rho]{<}K{>} & = & {<}{=}x\rho!{::}K{>} & \\
[\mathit{self}{=}x\rho]\Sigma x'{:}A_1.\, A_2 & = & \Sigma x'{:}[\mathit{self}{=}x\rho.1]A_1.\, [\mathit{self}{=}x\rho.2]A_2, & \quad \mathit{where}\ x \neq x' \\
\\
[\mathit{self}{=}x\rho]\alpha & = & \alpha & \\
[\mathit{self}{=}x\rho]\Pi x'{:}A_1.\, A_2 & = & \Pi x'{:}A_1.\, A_2 & \\
[\mathit{self}{=}x\rho]\lambda\alpha{::}K.\, A & = & \lambda\alpha{::}K.\, A & \\
[\mathit{self}{=}x\rho]A_1\, A_2 & = & A_1\, A_2 & \\
[\mathit{self}{=}x\rho]{<}{=}A{::}K{>} & = & {<}{=}A{::}K{>} & \\
[\mathit{self}{=}x\rho]x'\rho'! & = & x'\rho'! & \\
[\mathit{self}{=}x\rho]\mathbf{rec} & = & \mathbf{rec} & \\
[\mathit{self}{=}x\rho]\mathbf{ref} & = & \mathbf{ref} &
\end{array}
$$

Bibliography

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] Richard J. Beach. Experience with the Cedar programming environment for computer graphics research. Technical Report CSL-84-6, Xerox Corporation, Palo Alto, July 1985.

[4] Edoardo Biagioni. A structured TCP in Standard ML. In *Sigcomm '94*, London, England, August/September 1994.

[5] Edoardo Biagioni, Robert Harper, and Peter Lee. Standard ML signatures for a protocol stack. Technical Report CMU–CS–93–170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993. (Also published as Fox Memorandum CMU–CS–FOX–93–01).

[6] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Conference on LISP and Functional Programming*, Orlando, Florida, June 1994.

[7] Hans-Jürgen Böhm, Alan Demers, and James Donahue. An informal description of Russell. Technical Report 80–430, Computer Science Department, Cornell University, Ithaca, New York, 1980.

[8] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Kahn et al. [28], pages 1–50.

[9] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988. ACM.

[10] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[11] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, DEC Systems Research Center, Palo Alto, CA, March 1990.

[12] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[13] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. Early experience with Mesa. Technical Report CSL-76-6, Xerox Corporation, Palo Alto, October 1976.

[14] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduuction in typed λ-calculi. In *Seventh Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992.

[15] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[16] Robert Harper. Personal communication, 1993.

[17] Robert Harper. A propsoal for ML2000 (draft of december 16, 1994). (Unpublished), December 1994.

[18] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU–CS–94–116, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1994. (Also published as Fox Memorandum CMU–CS–FOX–94-02; to appear *Workshop on ML*, Orlando, FL, June, 1994.).

[19] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM, ACM.

[20] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-first ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[21] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML (version 3). Technical Report ECS–LFCS–89–81, Laboratory for the Foundations of Computer Science, Edinburgh University, May 1989.

[22] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[23] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[24] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU–CS–96–136R, Carnegie Mellon University, Pittsburgh, PA, 1996.

[25] James G. Hook. Understanding Russell: A first attempt. In Kahn et al. [28], pages 69–85.

[26] K. Jensen. *Pascal User Manual and Report.* Springer-Verlag, 1978. (2nd edition).

[27] Mark P. Jones. Using parameterized signatures to express modular structure. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 21–24, St. Petersburg Beach, FL, January 1996. ACM Press.

[28] Gilles Kahn, David MacQueen, and Gordon Plotkin, editors. *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science.* Springer-Verlag, June 1984.

[29] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-hall, Inc., 1978.

[30] Butler W. Lampson. A description of the Cedar language: A Cedar language reference manual. Technical Report CSL-83-15, Xerox Corporation, Palo Alto, December 1983.

[31] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*, pages 109–122. ACM, January 1994.

[32] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of POPL '95: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, CA, January 1995.

[33] Xavier Leroy. A modular module system. Research report 2866, INRIA, April 1996.

[34] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[35] Mark Lillibridge, Zhong Shao, and Robert Harper. Minutes of the ML2000 meeting at Portland, Oregon on January 16, 1994. Unpublished, 1994.

[36] Barbara Liskov, Russell Atkinson, et al. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[37] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[38] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming Languages and Systems — ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[39] David C. J. Matthews. POLY report. Technical Report 28, Computer Laboratory, University of Cambridge, 1982.

[40] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-78-1, Xerox Corporation, Palo Alto, February 1978.

[41] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[42] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[43] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Twelfth ACM Symposium on Principles of Programming Languages*, 1985.

[44] John Gregory Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. (Available as Carnegie Mellon University School of Computer Science technical report CMU–CS–95–226.).

[45] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[46] John Ophel. A polymorphic language with first-class modules. *Australian Computer Science Communications*, 17(1):422–430, February 1995.

[47] Amit Patel and Chris Stone. Minutes of the ML2000 meeting at King's Beach, CA. (Unpublished), August 1996.

[48] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[49] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.

[50] Benjamin Pierce. Bounded quantification is undecidable. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque*. ACM, January 1992.

[51] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1991.

[52] I. C. Pyle. *The Ada Programming Language*. Prentice-Hall International, 1981.

[53] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1987.

[54] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. Technical Report CSL-86-1, Xerox Corporation, Palo Alto, June 1986.

[55] Warren Teitelman. The Cedar programming environment: A midterm report and examination. Technical Report CSL-83-11, Xerox Corporation, Palo Alto, June 1984.

[56] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1983.