Dependently Typed Programming with Domain-Specific Logics

Daniel R. Licata

CMU-CS-11-105

February 28, 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Robert Harper, Chair
Karl Crary
Frank Pfenning
Greg Morrisett, Harvard University

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

This research was sponsored in part by the National Science Foundation under grant numbers CCF-0702381, CCR-0325808, CCR-0121633; by the US Army Research Office under grant number DAAD-190210389; and by the Pradeep Sindhu Computer Science Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: type theory, category theory, functional programming, dependent types, higher-dimensional type theory, abstract syntax, binding and scope, derivability, admissibility

With every day, and from both sides of my intelligence, the moral and the intellectual, I thus drew steadily nearer to that truth, by whose partial discovery I have been doomed to such a dreadful shipwreck: that man is not truly one, but truly two. . . . It was on the moral side, and in my own person, that I learned to recognise the thorough and primitive duality of man; I saw that, of the two natures that contended in the field of my consciousness, even if I could rightly be said to be either, it was only because I was radically both; and from an early date, even before the course of my scientific discoveries had begun to suggest the most naked possibility of such a miracle, I had learned to dwell with pleasure, as a beloved daydream, on the thought of the separation of these elements. If each, I told myself, could be housed in separate identities, life would be relieved of all that was unbearable; the unjust might go his way, delivered from the aspirations and remorse of his more upright twin; and the just could walk steadfastly and securely on his upward path, doing the good things in which he found his pleasure, and no longer exposed to disgrace and penitence by the hands of this extraneous evil. It was the curse of mankind that these incongruous faggots were thus bound together—that in the agonised womb of consciousness, these polar twins should be continuously struggling. How, then were they dissociated?

— Strange Case of Dr Jekyll and Mr Hyde*, Robert Louis Stevenson*

Abstract

This dissertation describes progress on programming with domain-specific specification logics in dependently typed programming languages. Domain-specific logics are a promising way to verify software, using a logic tailored to a style of programming or an application domain. Examples of domain-specific logics include separation logic, which has been used to verify imperative programs, and authorization logics, which have been used to verify security properties in security-typed languages. The first goal of the research described here is to show that it is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language. We demonstrate this fact with a significant new example, showing how to embed a security-typed language using dependent types.

This example suggests that better support for programming with logics in type theory will facilitate this style of program verification. The central notion in logic is consequence—entailment from premises to conclusions—and two notions of consequence are necessary for programming with logics: derivability, which captures uniform reasoning, and admissibility, which captures inductive proofs and functional programs. Presently, derivability is better supported in LF-based proof assistants, such as Twelf, Delphin, and Beluga, whereas admissibility is better supported in proof assistants based on Martin-Löf type theory, such as Coq, Agda, and Epigram. Our second contribution is to show that it is possible to implement, within a dependently typed programming language, a logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.

The above framework is simply-typed, which makes it suitable for programming with abstract syntax but not logical derivations. Our third contribution is to generalize this framework to dependent types, which we accomplish as an instance of a more general problem: We describe Directed Type Theory (DTT), a new notion of dependent type theory, inspired by higher-dimensional category theory, which equips each type with a notion of transformation on its elements. The structural properties of a logic arise as a special case, by considering a type of contexts equipped with an appropriate notion of transformation. DTT is an exciting development independently of our application, as it generalizes recent connections between type theory, homotopy theory, and category theory to the asymmetric case.

Acknowledgments

First and foremost, I would like to thank my advisor, Robert Harper. In one of our first meetings, Bob said "you should take a look at this work on indexed types," and seven years later, here we are. When you are picking an advisor, one of the things they tell you to think about is whether someone will be more hands-on or hands-off, and Bob has been both: I am very grateful for both all the collaboration—the afternoons spent working together on whiteboards, and the evenings at 61c—and the space—when I needed to figure things out for myself. Bob has an amazing eye for spotting when a solution is right, or when it can be improved. I would like to thank him for always pushing me to work harder and find the beautiful, canonical, solution to a problem—this has been essential to the work described here and to my development as a researcher.

Second, I would like to thank my internal committee members, Karl Crary and Frank Pfenning, both for their efforts on my dissertation and for many helpful conversations over the years. I have learned a lot from watching each of them solving problems. Third, I would like to thank my external committee member, Greg Morrisett, for his feedback on my dissertation, and for his work on Hoare Type Theory, which has been an inspiration for this work.

Outside of my committee, my closest collaborator was Noam Zeilberger, whose work on polarity and higher-order focusing was an essential tool for the early stages of this work. One afternoon, I came down to the lounge, found Noam sitting there, and said "I wonder if the LF function space is *positive*"—and several years of interesting and productive collaboration ensued. Noam has an unusual knack for taking seemingly impossible ideas and making them work out, and for paring something down to its essence.

Next, I would like to thank Jason Reed, Arbob Ahmad, and Jamie Morgenstern for collaborating on research described in this dissertation. One of my greatest joys in grad school was watching Jason do logic on a whiteboard. As undergraduates, Arbob worked on a project that refined my understanding of focusing, and Jamie worked on the security-typed programming example described below, which is the most practical motivation for the more theoretical work that follows it.

My work on directed type theory has benefited tremendously from conversations with Steve Awodey, Peter Lumsdaine, Chris Kapulkin, and Kristina Sojakova. I would also like to thank Vladimir Voevodsky for attracting my attention to higher-dimensional type theory, which turned out to be a very useful technique for attacking the problems considered in this dissertation.

The moment I decided to come to CMU was at the Saturday night dinner of the visit weekend, when, over beer at the Church Brew Works, Tom Murphy VII and

others explained the Curry-Howard interpretation of classical logic as continuations. This was not false advertising. I am extremely grateful for the many colleagues and friends in the PoP Group whose knowledge and company I have shared over the past seven years. My understanding of logic is almost entirely the product of many, many, long and delightful whiteboard conversations with Noam, Jason, Neel Krishnaswami, and Rob Simmons. Kevin Watkins introduced me to the wonders of focusing and canonical-forms-only presentations of type theory. Each of Kumar Avijit, David Baelde, Lars Birkedal, Kaustuv Chaudhuri, Derek Dreyer, Joshua Dunfield, Deepak Garg, Daniel Lee, Ruy Ley-Wild, William Lovas, Chris Martens, Sean McLaughlin, Tom Murphy VII, Susmit Sarkar, and Dan Spoonhower has contributed to this dissertation in some way.

I am also lucky to have had helpful conversations with many external colleagues in the proof theory, dependent types, and mechanized metatheory communities. I would especially like to thank Paul Levy, Thorsten Altenkirch, and Conor McBride for discussions about this work.

I would like to thank Simon Peyton Jones for a wonderful internship at Microsoft Research Cambridge, for getting me to think more about the practical applications of programming languages research, and for suggesting that working out a specific example of a domain-specific logic in more detail would be a good complement to working on general technology.

This dissertation would have been very different if not for the Agda proof assistant, and I would like to thank its developers, particularly Ulf Norell and Nils Anders Danielsson. I would also like to thank Wouter Swierstra for turning me on to Agda at ICFP'07.

I probably would not have gone into programming languages if not for the Schemeleh (that's Yiddish for "little Scheme") project in CS17-18 at Brown, both the year that I took the course, with John Hughes, and the years that I TAed it with Spike and with Philip Klein. And I certainly would not have gone into programming languages if not for several years of research with Shriram Krishnamurthi. I worked on software verification as an undergrad, and I remember telling Shriram that I was pretty sure I wanted to do programming languages, even though I hadn't really worked on it. His response was "yeah, it's what we do for beauty."

Grad school is hard, but rewarding, and when you're in a hole it's good to have people who know the way out: thanks Katrina, Jamie, Lorna, Lisi, and Julia. Otto and Radar, both of your parents have PhDs now, and we expect great things.

Finally, I would like to thank my parents, Ken and Sue, my brother, Matt, and my grandparents, Jean, Jules, Ruthe, and Gerry. When you're applying to grad school, they tell you not to tell your computer origin story in your personal statement. Fortunately, no one ever says that about your thesis document: A long time ago (I was so little that I don't even remember this story, except from being told about it), Dad brought home an Apple II from school. Mom and Dad set me up in the dining room playing Math Blaster or something and invited Grandma and Papa over. The trick worked, and pretty soon we had our own Apple IIGS in the house. Which is to say:

I'd like to thank my parents for setting me on a path to something that makes me happy, and for supporting me along the way.

Both of my grandmothers died while I was in grad school. One time, during my first or second year of grad school, Grandma Jean asked me how long the program took, and I told her that the average was five or six years. Her response, which surprised me in its frankness, was "That's too long! We won't be around to see you graduate." Technically, she made it: she died the evening of the day I sent out my dissertation draft.

Grandma Ruthe was the only one of my grandparents to ever use a computer, one that Matt and I built for her back when it was cool to order your motherboard, RAM, video card, etc., separately online. She kept records for her temple's sisterhood, checked her Juno e-mail, and played a bunch of virtual mahjong. Her brother, Uncle Alex, has a PhD in mathematics, and a copy of his dissertation was on the bookcase at Grandma and Papa's house. Grandma Ruthe used to joke that the only part of it that she understood was his name. I like to think that, here, she would have made it to page two.

Contents

List of Figures

Chapter 1

Introduction

Up until the fall of 2009, Carnegie Mellon University's computer science department was housed in a building named Wean Hall, an eight-story bunker of a building whose dominant architectural feature is bare concrete. On the fourth floor of Wean Hall was the graduate student lounge, a room decorated with a billowy sky-blue table, neon hanging lampshades, and floor-to-ceiling whiteboards. On the wall outside the lounge was a bronze plaque of eight quotations by Alan Perlis, the first head of Carnegie Mellon's computer science department, taken from his *Epigrams on Programming* (Perlis, 1982). Sixth from the top was the following:

> *It is easier to write an incorrect program than understand a correct one.*

This dissertation describes progress towards invalidating Perlis's epigram, in two ways: making incorrect programs harder to write and correct programs easier to understand.

To illustrate these ideas, imagine that it is the mid-1990s and you are writing the first airline search Web site, Ellipsez.com. Just after launching the first version, you notice that your customers tend to click through all the pages of search results looking for the flight with the lowest price. So, you think, Ellipsez.com should automatically sort the results by price, and display them from lowest to highest. To do this, you implement a function `sort` which takes a list of flights and puts them into increasing order by price. First of all, it is important that this new `sort` functionality does not crash: if it crashes, then Ellipsez.com will never display any results at all, so no one can buy anything through it. It is also important that the flights are actually ordered by price—if the best price is on page three instead of page one, your customers might miss it, and think that your competitors are searching more flights than you are. For similar reasons, it is important that `sort` does not drop any flights, or include flights that were not in the original list. These are examples of different notions of *correctness*.

At some point later, you, or maybe one of the many programmers you hire after Ellipsez.com gets popular, will notice that now the customers are clicking through the pages of search results looking for one that departs at a particular time, rather than just picking the cheapest one. So you want to go back and change the code for `sort` so that you can order flights not only by price, but also by time of departure, arrival, and so on. To do this, it is important that the code for `sort` be *understandable* so that you or other programmers can change and maintain it.

There are a variety of techniques that help programmers write correct and understandable software. In this dissertation, we focus on *static*, or *compile-time*, techniques, used when the

programmer is writing a program—as opposed to *dynamic*, or *run-time*, techniques, used when the user is running a program. One simple technique is *testing*: the programmer runs sort on some example search results and sees that it does the right thing. While testing can help find bugs and boost confidence, it cannot show that there is no circumstance in which sort fails. Proving that sort is correct on all possible inputs takes more sophisticated techniques based on *logic*. *Type systems*, as in programming languages such as ML (Milner et al., 1997) and Haskell (Peyton Jones, 2003), are a lightweight approach that provides certain limited but useful guarantees (for example, they rule out certain kinds of crashes). A more expressive, but also more costly, technique is to use a *specification logic* such as Hoare logic (Hoare, 1969), a mathematical formalism in which a program can be proved to have certain correctness properties ("for all lists l, sort produces a permutation of the flights in l which is in increasing order by price"). A *dependent type theory* (Martin-Löf, 1975) integrates these two approaches into a type system rich enough to express logical specifications. Type systems and specification logics also make programs more understandable. Because types and specifications provide a concise summary of the behavior of a piece of code, they are often easier to understand than the code itself. Additionally, because this documentation is machine-checkable, it does not get out of date. Moreover, types and specifications enable *modular* programming, where one piece of code relies only on the type or specification of another, allowing the two pieces of code to be modified independently.

However, different programs and correctness properties require different logics tailored to reasoning about them: For example, Hoare logic and separation logic (Reynolds, 2002) are used to reason about imperative programs which manipulate the state of memory in intricate ways. Temporal logics (Clarke et al., 2002) are used to reason about the evolution of concurrent processes. Authorization logics (Abadi et al., 1993) are used to verify security properties (Ellipsez.com does not leak your credit card number to anyone). Differential dynamic logic is used to reason about hybrid (discrete/continuous) cyber-physical systems (Platzer, 2010). Because different logics are appropriate for different tasks, specification logic design necessarily becomes part of the programming process.

What are the typical activities involved in designing a new logic? The first task is to define the logic, and write down the syntax of its propositions and proof rules. This must include a way of integrating the logic with the programming language, either by allowing logical formulas that refer to programs, as in a specification logic, or by externally assigning propositions to programs, as in a type system. To show that the logic is reasonable, it is common to prove some sort of correctness result about the logic: one may prove that the logic is consistent, or more generally that it ensures that programs have the properties of interest. Such proofs may use syntactic methods (structural induction on derivations), or semantic ones (showing that the logic is sound and/or complete with respect to a notion of truth). Next, to make verification practical, it is common to implement some sort of automated or interactive theorem proving for the logic.

At a high level, the first goal of the research described here is to advance the idea that

> *It is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language.*

Dependent types make logic design part of the programming process, so that programmers can define logics and use them both to reason about their code and to explain it to others. The result is *code that tells you why it works*. All of the tasks involved in programming with logics can be

carried out within a dependently typed host language, which improves on current practice, where it is common to implement logics and type systems either within language implementations or as external tools. Dependently typed languages provide richer tools for describing, studying, and implementing logics than the current simply-typed languages used in these implementations, and they permit program verification as well.

Though the particular approach that we advocate has some modern twists, this general approach to program verification is an old idea and it is only through decades of work on dependently typed proof assistants and programming languages that it is now becoming a reality. In Part I of this dissertation, we contribute some new examples as further evidence of the above claim. Our most significant new example shows how to do security-typed programming, in the style of the languages PCML5 (Avijit et al., 2010), Aura (Jia et al., 2008), and Fine (Swamy et al., 2010), within a dependently typed programming language. Second, we show how to represent and implement the semantics of Reed and Pierce (2010)'s type system for differential privacy (Dinur and Nissim, 2003; Dwork and Nissim, 2004; Dwork et al., 2006), which provides an extensible approach to implementing differentially private algorithms in a formally certified manner. Throughout the dissertation, we use the dependently typed programming language Agda (Norell, 2007).

Of course, we would like to achieve a state of affairs where it is not just possible, but *practical*, to program with logics using dependent types. At present researchers can, with some effort, do this type of programming, but we have not achieved the practicality necessary for wide-spread adoption. The remaining parts of this thesis make some technical contributions towards making programming with logics easier.

Mixing derivability and admissibility A key notion in logic is the *hypothetical judgement*, which codifies reasoning from assumptions. Any tool for programming with logics must support two notions of hypothetical judgement, *derivability*, which captures uniform reasoning, and *admissibility*, which captures inductive proofs and functional programs. These notions are essential for describing and programming with logics. Derivability is better supported in proof assistants that use type theory in the style of LF (Harper et al., 1993), whereas admissibility is better supported in proof assistants that use type theory in the style of Martin-Löf type theory (MLTT) (Martin-Löf, 1975).

Our aim is to heal this divide by proving better support for derivability in MLTT. In the process, we have an opportunity to go beyond previous work, by allowing admissibility and derivability to interact in new ways. Part II of this dissertation is devoted to studying these interactions. In particular, we show that

> *It is possible to implement, within a dependently typed programming language, a logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.*

We were originally motivated to consider these interactions by studying derivability and admissibility in the context of Zeilberger's higher-order focusing (Zeilberger, 2008a,b, 2009). Zeilberger's previous work showed how to do higher-order focusing for classical logic, and for the "positively-only" part of intuitionistic logic. Our work in this part also includes a formulation of higher-order focusing for full intuitionistic propositional logic, which is of interest independently

of the application.

Directed Dependent Type Theory The dependently typed analogue of the hypothetical judgement is called the generic judgement, and implementing a logical framework with genericity is not as easy. The difficulty comes in implementing the *structural properties* of the generic judgement. We analyze these difficulties as an instance of a much more general phenomenon, that of dependently typed programming with *directed types*.

In the standard interpretation of type theory, a type may be thought of as a *set* of values, and any two values may or may not be equal. This interpretation turns out to be level 1 in a hierarchy of more structured types. Down at level 0 are the *propositions*: proof-irrelevant types whose members are all equal (if there are any). Up at level 2 are types that contain values, but these values may be related in more ways than just equality. For example, one could have a type whose values are themselves (smaller) types, which are considered equal iff they are *isomorphic*—there are functions back and forth that compose to the identity. This generalizes the notion of a *quotient type* (a type equipped with an equivalence relation) by allowing multiple, computationally relevant proofs of equivalence. Level 3 gives the notion of *equivalence of categories*, or "isomorphism up to isomorphism," of category theory, and in general level n gives the (weak) n-equivalence of higher-dimensional category theory. Higher-dimensional type theories make this dimension hierarchy available to the programmer. The turnstile in typing judgements is interpreted as *functoriality*: any judgement $\Gamma \vdash J$ means not only that J is well-formed in Γ, but also that "equal" Γ's give "equal" J's, where "equal" is in the sense appropriate for the levels of Γ and J. These ideas are being explored in work connecting Martin-Löf type theory, higher-dimensional category theory, and homotopy theory (Awodey and Warren, 2009; Hofmann and Streicher, 1998; Lumsdaine, 2009; van den Berg and Garner, 2010; Voevodsky, 2010). This work promises a new class of proof assistants that intrinsically support proof irrelevance, equality, isomorphism, equivalence of categories, and so on, and thus will be much more convenient for formalizing mathematics.

The structural properties of the generic judgement are instance of higher-dimensional structure, given by entailment between logical propositions. However, entailment is not symmetric, and thus is not an instance of the higher-dimensional theories discussed above, which account only for symmetric notions of equivalence. This application prompted us to begin to study a new notion of *directed dependent type theory* which accounts for asymmetric or directed types. This requires not just a new semantic interpretation, but also a new proof theory, as MLTT proves symmetry of equivalence. In Part III, we study the 2-dimensional case, which accounts for types of level 2, and give it a semantics in category theory. We show that

> *A language with directed types provides a useful framework for describing the structural properties of the generic judgement.*

Directed type theory is of interest independently of our application, as it extends the connection between type theory, higher-dimensional groupoids, and homotopy theory to directed type theory, higher-dimensional categories, and directed homotopy theory.

Chapter 2

Background

2.1 Logical Frameworks

Two main tools are necessary to program with logics. The first is a mechanism for *representing* a logic as data in a program. This involves representing the abstract syntax of propositions and proofs. The second is a mechanism for *computing* with logics: writing recursive functions that case-analyze and traverse syntax. As we will see below, computation has many uses, such as proving meta-theorems about a logic, or implementing a proof search procedure.

2.1.1 First-order inductive definitions

In this thesis, we concentrate on the *proof-theoretic* study of logic, using natural deduction systems and sequent calculi (Gentzen, 1935). Proof theories are described using inductive definitions, presented as a collection of inference rules. For example, one may define a proposition to be true iff it is an element of the least set of propositions closed under a collection of inference rules of the form

$$\frac{A_1 \ \mathsf{true} \quad \ldots \quad A_n \ \mathsf{true}}{C \ \mathsf{true}}$$

We call such collections of rules *first-order inductive definitions*. First-order inductive definitions have a straightforward set-theoretic justification: the inference rules define a monotone operator, which has a least fixed point.

2.1.2 Hypothetical Judgements

While first-order inductive definitions suffice to define truth in an axiomatic, Hilbert-style, way, the most important notion in proof theory is *consequence*: entailment of a conclusion from an assumption. Consequence is used, for example, in the natural deduction (Gentzen, 1935) rule for implication introduction:

$$\frac{\dfrac{}{A \ \mathsf{true}}\,u \\ \vdots \\ B \ \mathsf{true}}{A \supset B \ \mathsf{true}}\,\supset I^u$$

This rule says that, to conclude that the proposition $A \supset B$ is true, it suffices to conclude the truth of B, using a new, scoped assumption that A is true. We indicate scoping by (a) labelling the assumption u, and (b) writing this label after the name of the inference rule, indicating that the rule *discharges* the assumption—the assumption may not be used below the rule, in another branch of the derivation tree.

What are the characteristics of assumptions? The first is *identity*: an assumption may be used to conclude what was assumed. That is, an assumption may be used as the leaf of a derivation tree; for example:

$$\dfrac{\dfrac{}{A\ \mathsf{true}}\,u}{A \supset A\ \mathsf{true}}\,\supset I^u$$

The second is α-*conversion*: derivations differing only by the labels of assumptions are considered equal. For example, the above derivation is the same as

$$\dfrac{\dfrac{}{A\ \mathsf{true}}\,v}{A \supset A\ \mathsf{true}}\,\supset I^v$$

The third is *substitution*: a derivation may be plugged in for an assumption. That is, given

$$\begin{array}{ccc} \mathcal{D} & & \dfrac{}{A\ \mathsf{true}}\,u \\ A\ \mathsf{true} & \text{and} & \mathcal{E} \\ & & B\ \mathsf{true} \end{array}$$

we can form a derivation

$$\begin{array}{c} [\mathcal{D}/u]\mathcal{E} \\ B\ true \end{array}$$

by replacing each leaf labelled u in $\mathcal{E}$ with $\mathcal{D}$.

This notion of assumptions is used pervasively in natural deduction. For example, in disjunction elimination

$$\dfrac{A \vee B\ \mathsf{true} \quad \begin{array}{c}\dfrac{}{A\ \mathsf{true}}\,u \\ \vdots \\ C\ \mathsf{true}\end{array} \quad \begin{array}{c}\dfrac{}{B\ \mathsf{true}}\,v \\ \vdots \\ C\ \mathsf{true}\end{array}}{C\ \mathsf{true}}\,\vee E^{u,v}$$

each branch has a scoped assumption, of A in one and of B in the other. Sequent calculi can be described using this notion of assumption as well (Pfenning, 1994).

The above rules can be rephrased in a one-dimensional syntax by collecting the assumptions into a *context* Γ. For example:

$$\dfrac{\Gamma, u : A\ \mathsf{true} \vdash B\ \mathsf{true}}{\Gamma \vdash A \supset B\ \mathsf{true}}\,\supset I^u$$

Then we can write the identity and substitution properties as follows:

$$\dfrac{}{\Gamma, u : J, \Gamma' \vdash J}\,u \qquad \dfrac{\Gamma, \Gamma' \vdash J_1 \quad \Gamma, u : J_1, \Gamma' \vdash J_2}{\Gamma, \Gamma' \vdash J_2}\,\mathsf{subst}$$

Here we write J to stand for an arbitrary judgement.

The one-dimensional notation reveals several additional properties that were tacit above:

$$\frac{\Gamma, \Gamma' \vdash J'}{\Gamma, u : J, \Gamma' \vdash J'}\ \mathsf{weakening} \qquad \frac{\Gamma, u_2 : J_2, u_1 : J_1, \Gamma' \vdash J'}{\Gamma, u_1 : J_1, u_2 : J_2, \Gamma' \vdash J'}\ \mathsf{exchange}$$

$$\frac{\Gamma, u_1 : J, u_2 : J, \Gamma' \vdash J'}{\Gamma, u_1 : J, \Gamma' \vdash J'}\ \mathsf{contraction}$$

Weakening says that an assumption may go unused. *Exchange* says that the order of assumptions does not matter. *Contraction* says that an assumption may be duplicated (note that it is an instance of substitution, where one assumption is plugged in for another). We refer to these five properties collectively as the *structural properties*, and we say that a judgement that obeys them is *structural*, or *a hypothetical judgement* (Martin-Löf, 1996).

As in the two-dimensional notation discussed above, we identify derivations that differ only in the labels of assumptions, such as

$$\frac{}{\Gamma, u : J, \Gamma' \vdash J}\ u \qquad \frac{}{\Gamma, v : J, \Gamma' \vdash J}\ v$$

This allows us to tacitly maintain the invariant that all assumptions are distinct; e.g. in $\supset I$, u can and must be distinct from all assumptions already in Γ.

2.1.3 Generic Judgements

A close relative of the hypothetical judgement is the *generic judgement* (Martin-Löf, 1996), in which an assumption may be used not only in the derivation tree, but also in the subjects of the judgement. This accounts for the notion of *variables*, or *binding and scope*, used pervasively in logic and programming languages. An example is the universal quantifier proposition $\forall x.A$, where the variable x is bound in the proposition A. The rules for this connective are as follows:

$$\frac{\Gamma, x : \mathsf{indiv} \vdash A\ \mathsf{true}}{\Gamma \vdash \forall x.A\ \mathsf{true}}\ \forall I \qquad \frac{\Gamma \vdash \forall x.A\ \mathsf{true} \quad \Gamma \vdash e : \mathsf{indiv}}{\Gamma \vdash [e/x]A\ \mathsf{true}}\ \forall E$$

We use the judgement $x : \mathsf{indiv}$ to mean that x is an *individual* or *term* of the first-order logic. The premise of the introduction rule is a derivation *generic in* the assumption x. The generic judgement satisfies structural properties analogous to above. For example, identity allows a variable to be used to form a term, while substitution of a term for a variable, $[e/x]e'$, is used in the $\forall E$ rule.

Using generic judgements to model variables brings us to another one of Perlis's epigrams (Perlis, 1982):

> *As Will Rogers would have said, "There is no such thing as a free variable."*

When modeled using generic judgements, every variable is bound, either in a term (as in $\forall x.A$) or in the context of a judgement (as in $\Gamma, x : \mathsf{indiv} \vdash A\,\mathsf{true}$). Treating the context as a binding site means that judgements can be *horizontally α-converted*, renaming variables in both the context and the right-hand side. For example, we equate the judgements $x : \mathsf{indiv} \vdash A\ \mathsf{true}$ and $y : \mathsf{indiv} \vdash [y/x]A\ \mathsf{true}$.

We call this approach to variables *pronominial*: every variable is a pronoun that refers to a designated binding site. Other approaches to variables, such as nominal logic (Pitts, 2003), instead conceive of variables as nouns, which exist independently of any scope.

The structural properties of the generic judgement are as follows:

$$\frac{}{\Gamma, x:\tau, \Gamma' \vdash x:\tau}\ x \qquad \frac{\Gamma \vdash e:\tau \quad \Gamma, x:\tau, \Gamma' \vdash J}{\Gamma, [\Gamma'/e]x \vdash [J/e]x}\ \mathsf{subst}$$

$$\frac{\Gamma, \Gamma' \vdash J'}{\Gamma, x:\tau, \Gamma' \vdash J'}\ \mathsf{weakening} \qquad \frac{\Gamma, x_2:\tau_2, \tau_1:\tau_1, \Gamma' \vdash J'}{\Gamma, x_1:\tau_1, x_2:\tau_2, \Gamma' \vdash J'}\ \mathsf{exchange}$$

$$\frac{\Gamma, x:\tau, y:\tau, \Gamma' \vdash J'}{\Gamma, x:\tau, [x/y]\Gamma' \vdash [x/y]J'}\ \mathsf{contraction}$$

Substitution reveals a somewhat subtle staging issue: to state the rule of substitution, we require a substitution operation on the syntax of judgements J. This can be sorted out in various ways. One is to distinguish between *terms* $e:\tau$ (used to represent propositions A, judgements J, etc.) and derivations of judgements $\mathcal{D} :: J$ and define substitution for terms prior to substitution for derivations. An alternative, which avoids this distinction, is to first define a substitution operation on a raw syntax of (possibly ill-formed) derivations, and then to later show that this operation satisfies the formation rule given by subst. The same issue comes up in contraction, but not weakening and exchange, because we have presumed that they do not alter the syntax of a judgement—a presumption we will return to below.

2.1.4 A Tale of Two Consequence Relations

Thus far, we have considered the hypothetical and generic judgements as *interfaces*: a relation $R(\Gamma, A)$ is a hypothetical/general consequence relation if it satisfies the appropriate structural properties. There are two commonly used implementations of this interface, called *derivability* and *admissibility*. In a sense, derivability is the least (initial) consequence relation, whereas admissibility is the greatest (final) one.

Derivability To warm up, we illustrate these ideas for a logic specified by a collection of first-order inference rules, which we will now write in a linear notation as $J \leftarrow J_1, \ldots, J_n$. The derivability hypothetical judgement is defined inductively to be the least relation closed under the following rules:

$$\frac{}{\Gamma, u:J, \Gamma' \vdash J}\ u \qquad \frac{J \leftarrow J_1, \ldots, J_n \text{ is a rule} \quad \Gamma \vdash J_1 \quad \ldots \Gamma \vdash J_n}{\Gamma \vdash J}$$

This definition generates the following induction principle:

For all predicates $P(\Gamma, J)$, if

- $P(\Gamma, J)$ whenever $u:J \in \Gamma$
- for all rules $J \leftarrow J_1, \ldots, J_n$, if $P(\Gamma, J_1)$ and ... and $P(\Gamma, J_n)$ then $P(\Gamma, J)$

then for all Γ and J, if $\Gamma \vdash J$, then $P(\Gamma, J)$.

It is simple to see that this definition of derivability defines a hypothetical judgement: identity is taken as a rule, and the remaining properties can be proved by induction.

Derivability, as specified by this inductive definition, is a very restricted notion of entailment: all you can do with an assumption is to slot it in at the leaf of a derivation tree. A consequence is that derivability is *open-ended* with respect to future extensions of the logic: if $\Gamma \vdash J$ with a particular collection of inference rules, then the entailment holds in any larger set of inference rules. A particularly illustrative case is a derivability $\mathsf{false} \vdash J$, where false is a judgement with no inference rules concluding it. Derivability does not satisfy the *principle of explosion*, which states that false entails any judgement J: there is no way to slot in an assumption of false at a leaf of a derivation tree to prove J (unless J happens to explicitly include an inference rule allowing this).

Admissibility Our second notion of consequence, admissibility, does not share this open-ended character. An admissibility $\Gamma \vDash J$ is proved by giving any function that, for all proofs of every assumption in Γ, delivers a proof of J. These functions are drawn from the *meta-logic*, the ambient logic in which we describe inference rules. For example, when working on paper it is common to take the meta-logic to be set theory. At this point, we do not specify the meta-logic precisely, but we do assume that a meta-logic function may be defined by induction on the proofs of the derivability judgement. We also assume that the meta-logic functions satisfy the structural properties, so that admissibility does.

Admissibility has many uses. For example, in sequent calculus, the proof rules are typically defined so that *cut* (the rule that allows the use of lemmas) is not derivable, but, crucially, is admissible: Cut-free sequent calculi guide the implementation of proof search and make certain meta-theoretic properties, such as consistency, obvious. On the other hand, a logic does not make sense unless proofs can in some sense be substituted for assumptions. Thus, it is typical not to include cut as an inference rule, but to prove that all instances of it hold. This theorem can be phrased as an admissibility, and proved by induction on derivations:

$$(A\ \mathsf{left} \vdash C\ \mathsf{right}), A\ \mathsf{right} \vDash C\ \mathsf{right}$$

The judgements $A\ \mathsf{left}$ and $A\ \mathsf{right}$ represent the side of the sequent on which a proposition is located (Pfenning, 1994). Note that cut is stated as a *higher-order admissibility*, in the sense that one of the assumptions is itself a derivability.

It is also useful, if less common, to use admissibility in the premises of inference rules. For instance, admissibility can be used to negate a judgement J, by writing $J \vDash \mathsf{false}$. As an example, consider a rule from the operational semantics of a language with mutable state:

$$\frac{l_1 = l_2 \vDash \mathsf{false} \quad \mathsf{lookup}(M, l_1) = v}{\mathsf{lookup}(M[l_2 \mapsto \_], l_1) = v}$$

When the memory is M extended with location l_2 pointing to some value, and l_1 is not equal to l_2, the result of lookup is the result of looking l_1 up in M. If the memory may contain duplicate bindings of a location, the disequality is necessary to ensure determinism of the operational

semantics. Such negated judgements are often written as informal *side conditions* on inference rules; with admissibility, they can be expressed as honest premises.

Another well-known example uses the admissibility generic judgement, which we will write $x : \tau \vDash J$. Such an admissibility is witnessed by a meta-logic universal quantifier. This notion underlies the ω-rule for natural numbers (Hilbert, 1931)

$$\frac{t : \mathsf{nat} \quad P(0)\ \mathsf{true} \quad P(1)\ \mathsf{true} \quad \ldots}{P(t)\ \mathsf{true}}$$

This rule states that to prove $P(t)$ it suffices to show $P(0)$, $P(1)$, and so on—the rule has infinitely many premises. This can be represented by an admissibility judgement:

$$\frac{t : \mathsf{nat} \quad n : \mathsf{nat} \vDash P(n)}{P(t)\ \mathsf{true}}$$

A witness for the second premise is a function that assigns each natural number n to a proof of $P(n)$, which formalizes the ellipsis above. We will see additional uses of admissibility premises in the formulation of logic described in Chapter 6.

From a programming point of view, admissibility corresponds to *computing with logics*. For example, a function which translates one logic to another, or implements a theorem prover, can be represented using $\vDash$ and programmed using functions that recursively transform syntax and derivations.

Relationship between derivability and admissibility. Every derivability implies an admissibility, but not vice verse. Assuming $\Gamma \vdash J$, we prove $\Gamma \vDash J$ by giving a function from closed derivations of Γ to closed derivations of J. But this is exactly the substitution property for $\vdash$. Thus, we say that every derivability determines an admissibility by substitution. On the other hand, returning to the example above, admissibility does satisfy explosion: the trivial function with no cases witnesses $\mathsf{false} \vDash J$. This is, of course, not stable under extension of the logic, because if we add an inference rule concluding false, the admissibility will no longer be valid. This shows that not every admissibility is also a derivability.

More generally, we define a class of consequence relations $\Gamma \to A$ such that

- $\to$ is *structural*: it satisfies identity, weakening, exchange, contraction, and substitution
- $\to$ is *closed*: $\Gamma \to J$ if there is a rule $J \leftarrow \Gamma$
- $\to$ is *complete*, in the sense that it does not add any theorems relative to derivability: if $\cdot \to J$ then $\cdot \vdash J$.

Next, we order consequence relations by containment: $\to \leq \to'$ iff for all Γ,J, $\Gamma \to J$ implies $\Gamma \to' J$.

The following theorem shows that derivability and admissibility are the least and greatest consequence relations:

PROPOSITION 2.1.1. *For all* $\to$, $\quad \vdash \leq \to \leq \vDash$.

Proof. For the first inequality, we can interpret the inference rules using structurality and closure under the rules. For the second, given $\Gamma \to J$ and a closed derivation of $\cdot \vdash \Gamma$, we must create a

closed derivation of $\cdot \vdash J$. By minimality of $\vdash$, $\cdot \to \Gamma$. Then we can use the structural properties to conclude $\cdot \to A$, which gives $\cdot \vdash A$ by completeness. □

2.1.5 Higher-order Inductive Definitions

Our warm-up definition of the derivability judgement above covered only first-order inference rules of the form $J \leftarrow J_1, \ldots, J_n$. However, in the examples, we used derivability and admissibility in the premises of inference rules. When do such premises make sense?

Admissibility premises Admissibilities may be problematic, as a rule like

$$\frac{J \vDash \mathsf{false}}{J}$$

does not induce a monotone operator: Consider building up the collection of proofs in stages, ranked by their size. At the first stage, J is vacuously true, so the premise holds. But then, at the second stage, J has a proof, so this proof is no longer valid! Thus, not all inference rules with admissibility premises give rise to an inductive definition. However, a rule such as

$$\frac{J \vDash \mathsf{false}}{K}$$

is permissible if J is in some sense defined prior to K: then it is permissible to inductively analyze the proofs of J in the definition of K. An example is the lookup rule given above, where equality of locations can be defined prior to lookup. This observation is formalized in Martin-Löf's theory of *iterated inductive definitions* (Martin-Löf, 1971) and the notion of *stratification* (Chan, 1988).

Derivability premises On the other hand, derivability can always be used in the premise of an inference rule. Intuitively, this is because derivability is such a restricted notion of consequence that it does not circumscribe its assumptions. Thus, it is unproblematic to use derivability assumptions of a judgement in its own definition. We can formalize this by reducing inference rules with derivability premises to first-order inductive definitions. That is, we specify, using first-order inductive definitions, a formalism that allows inference rules with derivability premises. These *second-order inference rules* have the form $J \leftarrow (\Gamma_1 \vdash J_1), \ldots, (\Gamma_n \vdash J_n)$. *Second-order derivability* is defined as follows:

$$\frac{}{\Gamma, u : J, \Gamma' \vdash J} \qquad \frac{J \leftarrow (\Gamma_1 \vdash J_1), \ldots, (\Gamma_n \vdash J_n) \text{ is a rule} \quad \Gamma, \Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma, \Gamma_n \vdash J_n}{\Gamma \vdash J}$$

For example, $\supset I$ would be represented as a rule

$$A \supset B \ \mathsf{true} \leftarrow (A \ \mathsf{true} \vdash B \ \mathsf{true}).$$

which, when plugged into the above schema, yields the usual implication introduction rule.

When we allow only first-order inference rules, as above, the definition of derivability is *parametrized* by a context Γ: for each context Γ, there is an inductive definition of the judgement $\Gamma \vdash -$. However, it is *indexed* by the judgement J: the definition of $\Gamma \vdash J$ refers to $\Gamma \vdash J'$ for different J', so the entire family of inductive definitions must be defined simultaneously. When we consider second-order rules, Γ becomes an index as well: the definition of $\Gamma \vdash J$ refers to $\Gamma' \vdash J'$ for different Γ' and J'.[1]

Thus, the induction principle must be *contextualized*, in the sense that it simultaneously proves a property of derivations in different contexts:

DEFINITION 2.1.2: INDUCTION PRINCIPLE FOR SECOND-ORDER RULES. For all predicates $P(\Gamma; J)$, if

- $P(\Gamma; J)$ whenever $u : J \in \Gamma$
- for all rules $J \leftarrow (\Gamma_1 \vdash J_1), \ldots, (\Gamma_n \vdash J_n)$, if $P(\Gamma, \Gamma_1; J_1)$ and ... and $P(\Gamma, \Gamma_n; J_n)$ then $P(\Gamma; J)$

then for all Γ and J, if $\Gamma \vdash J$, then $P(\Gamma; J)$.

Rules of the form $J \leftarrow (\Gamma_1 \vdash J_1), \ldots, (\Gamma_n \vdash J_n)$ are *local*: they specify only the additions to the context in each premise. Consequently, such rules are *pure* (Avron, 1991): they apply equally well in any context. Because all rules are pure, second-order derivability can be proved structural generically, independent of the particular inference rules.

Alternatively, one might consider rules written in a *global* notation, which allow arbitrary constraints on the context. Such rules may be *impure*, in the sense that they may not apply in all contexts, which may invalidate the structural properties. An example of such a rule is ! introduction in linear logic (Girard, 1987), which applies only in a context made up entirely of assumptions of the form $!A$:

$$\frac{!\Gamma \vdash A \ \mathsf{true}}{!\Gamma \vdash !A \ \mathsf{true}}$$

As a consequence of this rule (among others), linear logic does not satisfy weakening.

Global rules are not the only source of impurity: admissibility premises also may introduce impurity, as they allow rules to inspect the ambient context. For example, consider a rule of the form $J \leftarrow (K \vDash \mathsf{false})$. Then J may be provable in a context Γ, if it happens that it is impossible to prove K from Γ. However, this derivation cannot be weakened to the context (Γ, K), in which K is provable by identity! The study of the impurities arising from admissibility is one contribution of this thesis.

Thus far, we have been informal about the representation of Γ and the names of assumptions. However, we would like an inference rule formalism that allows general, in addition to hypothetical, judgements, in inference rule premises—where, in $\Gamma_i \vdash J_i$, the assumptions in Γ_i may occur in J. Thus, a context extension Γ, Γ_i must ensure that the assumption names in Γ do not clash with those in Γ_i. On paper, it is common to work with the following interface: In an inference rule, the assumption names in Γ_i are treated as a binding site, and in induction, these assumption names can be tacitly α-converted—e.g. $P(\Gamma, u : J; K)$ is the same as $P(\Gamma, v : J; [v/u]K)$. Thus, when proving that $P(\Gamma, \Gamma_i; J_i)$ implies $P(\Gamma; J)$, one never runs into a situation where the premise

[1] Though both are indices, there is folklore terminology for a further distinction between Γ and J: Γ is a "Protestant" index, which means that it varies only in premises, but is fully general in the conclusion of each rule. On the other hand, J is a "Catholic" index, which means that it may be specialized in the conclusion of rules (like $\supset I$).

or conclusion has the "wrong" assumption names. We call this *induction modulo* α*-conversion*. However, it takes some care to provide this interface in a proof assistant, as we now discuss.

2.1.6 Logical Frameworks

The formalisms for first-order inductive definitions, derivability, and second-order derivability defined above are examples of *logical frameworks*: a logical framework is a meta-logic that makes it convenient to specify and reason about logics. For example, the formalism for second-order derivability permits one to represent a logic by inference rules, and reason about it by induction-mod-α. The framework provides operations, such as the structural properties, generically for all logics thus represented. While logical frameworks are helpful when reasoning informally on paper, they are especially useful for mechanized reasoning on a computer, because one cannot be sloppy about details. In this section, we review the support for inductive definitions, derivability, and admissibility in existing logical frameworks.

Many logical frameworks are based on type theory, and in particular dependent type theory. In such frameworks, base judgements are represented as type families. For example, the judgement $A\ \mathsf{true}$ is represented by a type family $\mathsf{true} : \mathsf{prop} \to \mathsf{type}$, where the type $\mathsf{true}\ A$ classifies derivation trees witnessing $A\ \mathsf{true}$.

Next, the notions of derivability and admissibility are represented as *function types*. For admissibility, this representation should be unsurprising: an admissibility $J \vDash K$ is witnessed by a meta-logic function from derivations of J to derivations of K.

For derivability, we view a proof from assumptions as a term with free variables. An open term with a designated free variable determines a function (and therefore an admissibility) by substitution. For example, a deduction

$$\dfrac{\dfrac{}{A\ \mathsf{true}}\,u}{\begin{array}{c}\vdots\\ B\ \mathsf{true}\end{array}}$$

is represented by a term $\mathcal{D}$ with free variable u. This determines a function $\lambda u.\mathcal{D}$, that, when applied to a derivation $\mathcal{E}$, returns the substitution instance $[\mathcal{E}/u]\mathcal{D}$. Such functions are often called *parametric* or *schematic* or *uniform*, as they simply slot their argument in for the variable, without analyzing them.

The term *higher-order abstract syntax* (Church, 1940; Harper et al., 1993; Pfenning and Elliott, 1988) has historically been used to refer to the representation of derivability using functions of a logical framework, though the term has sometimes also been used to refer to representing admissibility using functions of a logical framework (Zeilberger, 2008b). In this thesis, we will use the term to refer to derivability, unless it is qualified to indicate otherwise.

Next, we overview the support for derivabilities and admissibilities in various logical frameworks.

Church's theory of types

The use of functions to represent derivability dates back to Church's definition of higher-order logic in his theory of types (Church, 1940). The relevant aspects for our purposes are: Base types

are used to represent syntax, such as a type i for individuals and o for propositions. Functions of the type theory are used to represent variable binding; e.g. the universal quantifier is represented by a constant $\mathsf{all} : (\mathsf{i} \to \mathsf{o}) \to \mathsf{o}$. Why should $\to$ be read as derivability, not admissibility? The reason is that base judgements are represented as uninterpreted base types, and there are no rules for deconstructing base types. Thus, the function space of the type theory allows only schematic or uniform use of assumptions, not inductive analysis of them. So the type theory can been seen as a meta-logic with support for derivability but not admissibility.

LF

This representation style was extended to judgements in the LF logical framework (Harper et al., 1993), which uses a dependent, rather than simple, type theory as a representation language. With a dependently typed framework, one can represent not just syntax (i and o) but also judgements (A true) and derivations. Judgements are still represented using indexed base types, so the LF function space $A \to B$ represents derivability, not admissibility. Moreover, the LF dependent function space $\Pi\, x{:}A.\, B$ represents generic judgements. However, there is no notion of admissibility internal to the LF type theory.

Instead, admissibility is handled by defining a separate meta-logic for reasoning about LF terms. In this meta-logic, the terms of LF (and in particular the *canonical forms*, or β-normal η-long terms) are specified by an inductive definition, so one can induct over canonical forms, just as we inducted over derivations of second-order derivability above. This observation is exploited in two parts of the LF methodology: This first is in proving *adequacy*—that the formal representation of a logic in LF is isomorphic to the informal representation written on paper. The second is in proving "meta-theorems" about LF representations of logics. In our terminology, these metatheorems correspond to admissibilities. Various systems built around LF, such as Twelf (Pfenning and Schürmann, 1999), Delphin (Poswolsky and Schürmann, 2008), and Beluga (Pientka, 2008), formalize a such a meta-language, permitting machine-checked proofs of meta-theorems. However, the downside of this approach is that one cannot use admissibility in LF representations, which precludes some of the examples described above.

Schürmann et al. (2001) describe a different approach to integrating admissibility. In their language, a modal type $\Box A$ classifies closed terms of type A and is eliminated by primitive recursion; Despeyroux and Leleu (1999) describe a generalization with dependent types. Unlike the LF-based approaches described above, these languages are not syntactically stratified into separate representational and computational parts. Instead, they reuse the same arrow for derivability (e.g., a function $A \to B$ where A and B do not include $\Box$, is used for representation) and admissibility (e.g., a function $\Box A \to B$ can decompose A by primitive recursion). However, these languages do not permit admissibility functions such as $\Box A \to B$ in datatype definitions.

Martin-Löf Type Theory

Dependent types are also the basis for a variety of proof assistants/programming languages based on Martin-Löf type theory (Martin-Löf, 1975), such as NuPRL (Constable et al., 1986), Coq (Coq Development Team, 2009), Epigram (McBride and McKinna, 2004), Agda (Norell, 2007), Dependent ML (Xi and Pfenning, 1999), Ωmega (Sheard, 2004), and ATS (Xi, 2003). We will refer

to these, especially the mostly-pure languages like NuPRL, Coq, Epigram, and Agda, as *MLTT*s. MLTTs provide a richer collection of types than LF, such as inductively defined datatypes and indexed families of datatypes (Dybjer, 1991). Thus, a base judgement (such as true : prop → type) can be represented either as an inductive type or as a base type.

When syntax and judgements are represented as inductive types, the function space of the theory represents admissibility, not derivability: a function $J \to K$ may case-analyze or induct on the proof of J. Thus, one can straightforwardly encode inference rules with admissibilities as premises using iterated inductive definitions (Martin-Löf, 1971), or datatypes with higher-order premises. For example, the ω-rule is represented as follows (using Agda notation, but being informal about the representation of variables):

```
data _true : Prop → Set where
  omega :  ( (n : Nat) → (A [ n / x ]) true )
       → (all x.A) true
  ...
```

In Agda notation, naming the type family _true allows it to be used post-fix. The constructor omega says that, given a function that maps each natural number n to a proof of A[n/x], we can conclude allx.A. Because Nat is an inductive type, such a function may be defined by recursion.

On the other hand, we are cheating by assuming this notation for variables and substitution: when syntax and judgements are represented as inductive types, MLTTs do not provide any built-in support for derivability. For example, if we try to represent syntax using MLTT functions as follows:

```
data Indiv : Set where
  ...

data Prop : Set where
  all : (Indiv → Prop) → Prop
  ...
```

then Indiv → Prop represents admissibility, not derivability. In the literature, such representations are often said to have a problem with "exotic terms" (Despeyroux et al., 1995)—elements of the function space that do not correspond to the desired syntax. This is true if the desired syntax is in fact derivability. However, if the desired syntax is admissibility, then the exotic terms are not exotic at all, but exactly what you want. That is, there is nothing *wrong* with using admissibility in syntax; it is simply a different notion than derivability. For example, the above use of admissibility defines a syntax of propositions with induction over individuals. The same idea is exploited in Zeilberger's higher-order focusing (Zeilberger, 2008a,b, 2009), which is "higher-order" in this admissibility sense. These higher-order representations facilitate simple proofs of cut elimination for logics with inductive types, using iterated inductive definitions in the meta-logic.

As discussed above, admissibility requires an iteration/stratification condition, which precludes putting a judgement to the left of an admissibility in its own definition. For example, we might try to extend the above logic with a second-order quantifier, represented by a new datatype constructor all2 : (Prop → Prop) → Prop. However, this constructor is rejected by the positivity checker of a system like Agda, which checks stratification. Naïvely extending a type theory with general (non-stratified) recursive types permits the definition of non-terminating terms of every

type, which makes the type theory inconsistent as a logic. In the literature, this is often referred to as a problem with "negative occurrences" of a type in its own definition. While these negative occurrences rule out certain uses of *admissibility*, it is important to realize that they are not a problem for *derivability*—as argued above, there is no problem putting a judgement to the left of a derivability in its own definition. Thus, claims that "higher-order abstract syntax" (meaning derivability) runs into problems with negative occurrences are predicated on a confusion between derivability and admissibility. Similarly, claims that "you cannot induct over higher-order abstract syntax" apply to admissibility, but not to derivability—one can perfectly well induct over derivability, e.g. using the induction principle for second-order derivability described above.

Higher-order approaches to derivability An alternative to representing base judgements as inductive types is to represent them as base types (as in an LF representation). Then the function space of the type theory represents derivability, not admissibility. For example, one could represent propositions by parametrizing over the following variables:

$\mathsf{Indiv} : \mathsf{Set}$
$\mathsf{Prop} : \mathsf{Set}$
$\mathsf{all} : (\mathsf{Indiv} \to \mathsf{Prop}) \to \mathsf{Prop}$
$\mathsf{all2} : (\mathsf{Prop} \to \mathsf{Prop}) \to \mathsf{Prop}$
...

There is no requirement for stratification if Prop is a base type. However, this representation affords no support whatsoever for admissibility within MLTT. Most of the time, one represents a logic in a proof assistant not just to reason in the logic, but also to reason about it—to prove theorems or to write recursive functions.

The need for admissibility leads to an interesting twist on the above representation, called *weak higher-order abstract syntax* (Despeyroux et al., 1995): replace the occurrence of Prop on the left of the arrow by a base type PropVar, and let occurrences on the right of an arrow refer to an inductive type. Then functions $\mathsf{PropVar} \to \mathsf{C}$ represent derivability, but functions $\mathsf{Prop} \to \mathsf{C}$ represent admissibility. However, WHOAS runs into some problems as well: First of all, because the type on the left is different than the type on the right, it is necessary to define substitution explicitly. Moreover, many recursive functions over syntax, including substitution, require testing the equality of variables. However, as soon as you add, e.g., $\mathsf{varEq} : \mathsf{PropVar} \to \mathsf{PropVar} \to \mathsf{Bool}$, the function space $\mathsf{PropVar} \to \mathsf{C}$ no longer corresponds to derivability, as it allows a form of introspection of variables.

Several solutions to this problem have been considered: The first (Despeyroux et al., 1995) is to allow a meta-language type $\mathsf{PropVar} \to \mathsf{C}$ that is too broad, and then isolate the subset of such functions that represent derivabilities using a predicate valid. However, the proof of valid follows the structure of a term exactly. Thus, writing down a function f and its validity proof amounts to writing both a higher-order and first-order representation of the same term. Thus, this representation is only tenable in a system with enough proof automation that these obligations can be discharged automatically.

A second solution, taken in the theory of contexts (Bucalo et al., 2006; Miculan, 2001), is to avoid adding any operations that would create exotic terms, while supporting admissibility by postulating various operations on syntax. For example, decidability of variables is postulated at a

separate propositional level only, so that it cannot be used to define in a function. Other postulates include the existence of a variable that does not occur in a given term, and recursion/induction principles that fix the meaning of $\mathsf{PropVar} \to \mathsf{Prop}$. A downside of this approach is that it requires an extension of MLTT with these postulates as axioms. Moreover, the operational semantics of these axioms (e.g. the computational behavior of a recursor) can be given only by proofs of propositional equations, not as part of the definitional equality of the type theory.

A third solution, called parametric higher-order abstract syntax (Chlipala, 2008), involves mediating between an abstract and concrete type of variables. Rather than fixing PropVar to be a single base type, quantified at the outside of the program, one sometimes chooses PropVar to be a type variable, and sometimes instantiates it concretely, using polymorphic quantification over types. To accomplish this, the datatype of propositions is parametrized by the type of proposition variables, and in a polymorphic term of type $\forall\alpha.\mathsf{Prop}(\alpha)$ the functions $\alpha \to \mathsf{Prop}$ must be uniform, representing derivability. But when programming with such terms, the quantifier can be instantiated by a concrete type, e.g. nat, so that variables can be compared for equality. However, working with PHOAS terms is somewhat idiosyncratic. For example, proving the correctness of a translation from PHOAS to a well-scoped first-order representation requires parametricity properties of the meta-logic, which no current implementations of MLTT provide (alternatively, one can translate PHOAS terms equipped with a separate proof of well-formedness, but that proof is similar to a first-order representation).

One weakness of these three approaches is that they do not provide the contextualized induction principles described above, which permits proving a context-indexed family of properties $P(\Gamma, J)$ about a derivation of $\Gamma \vdash J$. This is because these approaches identify the object-logic context with the meta-logic context, so the proof of a meta-theorem is "inside" an object logic context Γ, and thus cannot talk about the entire context explicitly. This situation is similar to two of the LF-based systems, Twelf and Delphin, but different than Beluga. In Twelf and Delphin, context invariants (e.g. "every term variable has an associated typing derivation") can be maintained without mentioning the context explicitly—in Twelf using the `%worlds` declaration, and in Delphin using functions from a type of variables. The latter approach could be used with WHOAS, but not in the theory of contexts (defining the functions require case-analysis on variables). In PHOAS, one way to maintain such invariants is by instantiating the type of variables appropriately. However, some theorems require a more detailed statement about the context, which does not simply associate some invariant with each variable in isolation. For example, when proving a substitution theorem for first-order logic, the assumptions to the left of the variable being substituted for remain unchanged, whereas the assumptions to its right are substituted into. One approach to proving such theorems, which has been used in Twelf, is to define an alternative representation where contexts are explicit, and to mediate between this and the usual representation (Crary, 2008).

Another problem with these three approaches is that it is unclear how well they scale to assumptions of order greater than two. For example, in the theory of contexts, one adds an induction principle for $\mathsf{PropVar} \to \mathsf{Prop}$, allowing inductive analysis of such functions. However, this means that a function of type $(\mathsf{PropVar} \to \mathsf{Prop}) \to \mathsf{Prop}$ corresponds to admissibility, not derivability. One could imagine considering a base type PropVarFn representing derivability assumptions of such functions, but this has, to our knowledge, not been worked out.

These three solutions show that it is possible to model derivability using the function space of MLTT, but either with some proof overheads (for WHOAS) or by adopting a somewhat idiosyncratic programming model (in the the theory of contexts or PHOAS). In any case, these approaches do not provide direct support for explicit reasoning about contexts, or for higher-order assumptions.

First-order approaches to derivability An alternative to modeling derivability as MLTT functions with a particular domain type is to model it using first-order means. To do so, it is necessary to choose a representation for contexts and variables.

Named form. The most basic representation is to use explicit names: a binding site, in a context or in a term, is represented as a pair of a a name (e.g. a string) and a judgement (e.g. $\Gamma, (\text{"}u\text{"}, A\mathsf{true})$), and variables are represented by occurrences of the same name. The advantages of this representation are that it is easy to write down terms, and that it matches informal practice. A problem is that it requires quotienting by α-conversion, so that judgements that differ only by the names of assumptions are considered equal. Quotient types are not very well supported in existing proof assistants, and in informal practice it is common to elide the proofs that operations respect α-equivalence.

de Bruijn indices. A second possibility is to represent contexts as lists of judgements $(\Gamma, A\mathsf{true})$, and represent variables as indices into a list Γ. This representation is known as de Bruijn indices (de Bruijn, 1972). This gives unique representations of α-equivalence classes, but makes terms somewhat difficult to write down and work with, because the de Bruijn indices are harder to write and read than names. Another disadvantage of de Bruijn indices is that weakening and exchange modify a term, as they must replace one index with another. A third problem is that terms may contain dangling free variables, which do not correspond to any variable in scope.

This last problem can be solved by using *well-scoped de Bruijn indices* (Altenkirch and Reus, 1999; Bellegarde and Hook, 1994; Bird and Paterson, 1999): representing terms as a dependent type indexed by the context, so that dangling indices are ruled out by typing. For example:

```
data _∈_ : Prop → Ctx → Set where
  i0 : ∀ {Γ A}
     → A ∈ (A :: Γ)
  iS : ∀ {Γ A B}
     → A ∈ Γ
     → A ∈ (B :: Γ)

data _⊢_ : Ctx → Prop → Set where
  id : ∀ {Γ A}
     → A ∈ Γ
     → Γ ⊢ A
  ⊃I : ∀ {Γ A}
     → (A :: Γ) ⊢ B
     → Γ ⊢ (A ⊃ B)
```

We represent variables by a type $\mathsf{A} \in \Gamma$, which is a unary number indexed by a context and a proposition being judged to be in that context. Next, we represent derivability by a type family $\vdash$ indexed by a context Γ and a proposition being judged true in Γ. The rules for this type family

consist of identity id, as well as the inference rules for each connective. The representation is isomorphic to the intended on-paper definition of derivability, and gives you direct access to the appropriate contextualized induction principle. However, while the indexing makes the de Bruijn indices somewhat easier to manage (many incorrect manipulations of them are type errors), it is still less convenient than names.

Locally/globally named/nameless. Another possibility is to distinguish bound variables (references to a binding site in a term) from free variables (references to a binding site in a context), and to use a different representation for each (Aydemir et al., 2008; McKinna and Pollack, 1999). A common choice is *locally nameless*, which uses de Bruijn indices for bound variables but names for free variables, though different permutations of {locally,globally} {named, nameless} have been considered. In general, an advantage of distinguishing bound and free variables is that substitution need not worry about *capture*: when substituting $[M/x]N$, it is necessary to ensure that the free variables in M do not collide with the bound variables in M, which would cause those variables to refer to the wrong binding site. If bound and free variables are differentiated, then a free variable can never refer to a binding site in a term. However, the cost of this representation is that, when traversing a binder, it is necessary to *open* the term, replacing the bound variable with a free variable.

Regarding locally nameless / globally named specifically, one advantage relative to de Bruijn is that weakening and exchange of free variables do not change the term. However, while locally nameless gives unique representatives of α-equivalence classes of bound variables, it does not immediately allow horizontal α-conversion of judgements. To ensure that judgements can be α-converted, some care is required when writing inference rules. For example, in rules that introduce assumptions, it is common to mediate between existential and universal quantification over fresh variables:

$$\frac{\text{there exists } u \notin \Gamma \text{ such that } \Gamma, u : A \ \mathsf{true} \vdash B \ \mathsf{true}}{\Gamma \vdash A \supset B \ \ \mathsf{true}} \qquad \frac{\text{for all } u \notin \Gamma, \Gamma, u : A \ \mathsf{true} \vdash B \ \mathsf{true}}{\Gamma \vdash A \supset B \ \ \mathsf{true}}$$

The existential rule is easier to introduce, because it requires only that one show the premise for a particular fresh variable. However, it is harder to reason from, because one only knows the premise for some fresh variable, and in some proofs, this fresh variable might need to be matched up with another fresh variable introduced elsewhere. On the other hand, the universal rule is harder to introduce, but easier to reason from, because the inductive hypothesis will give the premise for all fresh variables. One technique for using locally nameless involves mediating between these two rules: for example, one might define the system with the universal rule, but then prove, using a notion of *permutation of variable names* as in nominal logic (Pitts, 2003), that the existential rule is admissible—if the premise holds for some fresh variable, then it holds for all of them. Another technique is to use cofinite quantification (Aydemir et al., 2008), which uses a universal quantifier, but rather than demanding that the variable be fresh for Γ, one demands that it be fresh for some existentially quantified set L. This technique sometimes avoids the need to prove the existential rule admissible.

A disadvantage of these techniques that distinguish bound and free variables is that, unlike well-scoped de Bruijn indices, the scoping of free variables is not represented intrinsically in a term's type. For example, a locally nameless syntax will be typically have a term constructor $\mathsf{fvar} : \mathsf{Name\ A} \to \mathsf{Term\ A}$ allowing a term to be constructed from a name. However, this provides

no information about which names may appear in a term. Moreover, as soon as you try to index Term by a context of free variables, and check that all names are in this context, you end up back at a de Bruijn representation, because the proof that a name is in a context is exactly a de Bruijn index. Thus, locally nameless works best in a proof assistant with good support for automation, where a $\mathsf{Term}\ \Gamma\ A$ can be represented as a $\mathsf{Term}\ A$ paired with a separate proof that all of its free variables are in Γ, and these proof obligations can be discharged automatically by the proof assistant.

Structural properties. A disadvantage of all of these first-order representations, when used directly, is that the structural properties must be proved for each datatype. It would be nicer to instead have a logical framework that guarantees that the structural properties hold. Several tools, Ott (Sewell et al., 2007), Lambda Tamer (Chlipala, 2007), and LNGen (Aydemir and Weirich, 2010), provide frameworks for writing down definitions of syntax with binding, and provide generic implementations of the structural properties. However, a disadvantage of these approaches is that they are external to the type theory, implemented as tactics or stand-alone tools. A second disadvantage is that the notion of datatype with binding does not allow full exploitation of the host language—for example, they do not allow admissibility functions of the host language to be used in syntax.

Another approach to presenting the structural properties, taken in Hybrid (Ambler et al., 2002; Capretta and Felty, 2007; Momigliano et al., 2007) and Barzilay and Allen (2002); Hickey et al. (2006)'s work, is to define derivability using an underlying de Bruijn representation, but then to present a higher-order interface to it, using the admissibility functions of the host theory. As with WHOAS, this involves defining a predicate that identifies the subset of $\mathsf{Prop} \to \mathsf{Prop}$ functions that corresponds to derivability—those that, when applied, act like substitution into some de Bruijn term. Like WHOAS, this approach also requires managing some proof obligations that are external to a term. However, it improves on WHOAS by providing substitution, in addition to weakening, contraction, and exchange, as function application.

Logic Programming

The integration of derivability and admissibility has also been studied in the context of logic programming, as opposed to functional programming (Gacek et al., 2008; Miller and Tiu, 2003). Unlike LF, their approach distinguishes *terms* (the subjects of propositions) from propositions. The term level is a higher-order λ-calculus, in which derivability can be represented using higher-order abstract syntax. The propositional level includes two quantifiers, $\forall x.A$ and $\nabla x.A$, corresponding to admissibility and a notion of derivability, respectively (the exact structural properties that ∇ satisfies differ in different treatments of the connective). This allows admissibility and derivability generic judgements to be mixed at the propositional level, but not in terms. Like Twelf and Delphin, but unlike Beluga, in admissibilities the object-language context is identified with the generic judgement assumptions in the meta-language context.

Another logic programming idea that is relevant to the present work is definitional reflection (Hallnäs, 1991; Schroeder-Heister, 1993), which supposes a database of rules used for both building proofs of propositional atoms and for deriving consequences of atoms by "reflection" (i.e., by inverting the rules). Through the Curry-Howard interpretation, the rule database corresponds to a database of datatype constructors, which can be used both to build datatype values

and to define functions by pattern-matching. Derivability can be thought of as an extension of definitional reflection which permits this database of rules to *vary,* by introducing a *scoped datatype constructor.*

Functional Programming

Miller (1990) describes an early account of mixing derivability and admissibility. This takes the form of an extension to ML with a new type `'a => 'b` representing a term of type `'b` with abstracted parameter `'a`, as well as a restricted form of higher-order pattern-matching. In this proposal, the domain `'a` must be not only an equality type, but also a user-defined datatype, since the meanings of base types such as `int` or `string` should not be open-ended. This restriction corresponds to separating out a class of base judgements that may be hypothesized, a technique we will borrow in Chapter 5. Moreover, the codomain `'b` must be an equality type, which precludes making a derivability assumption in an admissibility (as ML functions are not equality types)—this avoids many of the complications that we will address below.

Nominal Logic

Nominal logic (Gabbay, 2010; Pitts, 2003) is another technique used to represent derivability judgements. However, nominal logic is both *more general than* and *not necessary to represent* derivability. Gabbay explains nominal logic as follows (Gabbay, 2010):

> If there is a single idea behind nominal techniques, it is to let names inhabit a denotation directly as a form of data ... That is, the 'x' in $\lambda x.r$, $\forall x.\varphi$, $\int_x f(x)dx$, and $\nu x.P$ has an independent denotational reality. This x is, in a mathematical sense that we will make formal, a 'real thing': a name.

Nominal logic studies names as *nouns*, which have meaning independent of any scope. On the other hand, derivability assumptions are *pronouns* that serve only as a reference to a binding site. It is possible to implement derivability using the names of nominal logic, in the same way that it is possible to implement a named representation from scratch—except the nominal framework provides notions of name permutation and α-equivalence, which simplifies the task considerably. However, it is not necessary to think of derivability in this way, any more than it is necessary to think of it in terms of de Bruijn indices. Thus, we view nominal logic as one potential implementation technique for derivability. That said, nominal logic is more general than derivability, and the notion of a name as a real thing is sometimes useful. For example, memory locations in an operational semantics for a language with state can be fruitfully modeled as names.

In one respect, names are easier to integrate with admissibility than derivability is: in nominal languages such as FreshML (Shinwell et al., 2003), the type of names is kept open-ended (it is considered to have infinitely many inhabitants). Thus, any admissibility function on syntax with binding must account for arbitrarily many names, and is therefore weakenable. However, many functions on syntax are only defined for certain classes of contexts (e.g., only closed arithmetic expressions can be evaluated to a numeral), and the nominal approach does not allow these invariants to be expressed in a program's type.

However, in another respect, names are more difficult to integrate with admissibility: when names are things, it takes some work to ensure that admissibility functions respect α-equivalence. There are several approaches to this problem. Static approaches, such as the type system in Pitts and Gabbay (2000) and and the specification logic in Pure FreshML (Pottier, 2007) use machinery for ensuring that certain names are fresh with respect to (roughly "not free in") certain computations. Using this machinery, one can state and prove a *freshness condition for binders* (Pitts, 2006) for each function definition, demonstrating that the bound variables are fresh for the the result. Pitts and Gabbay (2000) employ a conservative freshness analysis to discharge these conditions. Pottier (2007) describes a specification logic in which these conditions can be proved. A dynamic approach, proposed by Shinwell et al. (2003), exploits an effectful operational semantics to ensure that the conditions cannot be violated.

2.2 Agda

We briefly review Agda's syntax, referring the reader to the Agda Wiki (`http://wiki.portal.chalmers.se/agda/`).

Dependent functions Dependent function types are written as $(\mathsf{x} : \mathsf{A}) \to \mathsf{B}$. An implicit dependent function space is written $\{\mathsf{x} : \mathsf{A}\} \to \mathsf{B}$ or $\forall\ \{\mathsf{x}\} \to \mathsf{B}$. Arguments to implicit functions are inferred by default. They can also be explicitly instantiated positionally by writing $\mathsf{f}\ \{\mathsf{a}\}$ or by name by writing $\mathsf{f}\ \{\mathsf{x} = \mathsf{a}\}$. Non-dependent functions are written $\mathsf{A} \to \mathsf{B}$.

Anonymous functions are written $\lambda\ \mathsf{x} \to \mathsf{e}$. Named functions are defined clausally by pattern matching:

```
append : {A : Set} → List A → List A → List A
append [] ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

Let bindings are written as **let** x = e1 ... **in** e.

Infix and Mixfix Operators Infix operators are declared with underscores:

```
_++_ : {A : Set} → List A → List A → List A
l1 ++ l2 = append l1 l2
```

Mixfix operators are also allowed; e.g. a function if_then_else can be used as if e then e1 else e2.

Predicative Hierarchy Set is the classifier of classifiers in Agda. Set is a synonym for Set Z, where Z : Level is a universe level. Sets are stratified into universe for size reasons, to avoid Set : Set. Agda supports explicit universe polymorphism using quantification over Level.

Inductive Types Inductive types are defined by a datatype notation:

```
data List {l : Level} (a : Set l) : Set l where
  [] : List a
  _::_ : a → List a → List a
```

Non-uniform inductive families are allowed. For example, dependent de Bruijn indices are represented as follows:

```
data _∈_ {A : Set} : A → List A → Set where
  i0 : {x : A} {xs : List A} → x ∈ (x :: xs)
  iS : {x y : A} {xs : List A} → y ∈ xs → y ∈ (x :: xs)
```

For any Set A, and terms x and xs of type A and List A, there is a type x ∈ xs. The first constructor, i0, creates a proof of x ∈ (x :: xs)—i.e. x is the first element of the list. The second constructor iS, creates a proof of x ∈ (y :: xs) from a proof that x is in the tail.

As a simple example of dependent pattern matching, we define an n-ary version of iS:

```
skip : {A : Set} (xs : List A) {ys : List A} {y : A}
   → y ∈ ys → y ∈ (append xs ys)
skip [] i = i
skip (x :: xs) i = iS (skip xs i)
```

The fact that this code type-checks depends on the computational behavior of append; e.g., in the first case, the expression append [] ys reduces to ys, so we can return the index i unchanged.

Many basic type constructors can be defined as datatypes: we write Either A B for sums, Bool for booleans, Void for the empty type, Σ A B, where A : Set and B : A → Set, for dependent pairs, and A × B for non-dependent pairs.

Agda allows overloading of datatype constructors between different types. As a convention, we overload ▷ and ▶ for injections from one type to another.

Intensional Equality Intensional equality can be defined by a datatype

```
data Id {l : Level} {A : Set l} : A → A → Set l where
  Refl : {a : A} → Id a a
```

This notion of equality is propositionally proof-irrelevant: we can prove that any two terms of type Id x y are themselves Id. It is not functionally extensional, in the sense that one cannot prove

```
ext : ((x : A) → Id (f x) (g x)) → Id f g
```

It is common to postulate this as an axiom, though.

Universes A universe is specified by an inductive datatype of *codes* for types, along with a function mapping each code to a Set. For example, a simple universe with an empty type, a unit type, and binary products is specified as follows:

```
data Type : Set where
  ‘0 : Type
  ‘1 : Type
  _⊗_ : Type → Type → Type
Element : Type → Set
Element‘0 = Void
Element‘1 = Unit
Element (τ1 ⊗ τ2) = (Element τ1) × (Element τ2)
```

In the right-hand side of Element, we write A × B for the Agda pair type, etc.

Datatype-generic programs are implemented by recursion over the codes; e.g, every element of the universe can be converted to a string:

```
show : (τ : Type) → Element τ → String
show‘0 ()
show‘1 <> = "<>"
show (τ1 ⊗ τ2) (e1, e2) =
  "< " ++ (show τ1 e1) ++ " , " ++ (show τ2 e2) ++ " >"
```

In the first clause, the empty parentheses are a refutation pattern, telling Agda to check that the type in question (in this case Element‘0) is uninhabited, and allowing the programmer to elide the right-hand side.

As another example, we will often view booleans as a two-element universe, with only True inhabited:

```
data Bool : Set where
  True : Bool
  False : Bool
Check : Bool → Set
Check True = Unit
Check False = Void
```

Because Agda implements extensionality for Unit (there is only one record with no fields), terms of type Check True can be left implicit and inferred.

With The **with** construct allows a new column to be added to a pattern-matching function definition. This has two uses: the first is simple subsidiary case analysis, which would be written with case in ML or Haskell. For example:

```
forgetMaybe : {A : Set} {B : A → Set} → ((x : A) → Maybe (B x)) → (A → Bool)
forgetMaybe f x with f x
... | Some _ = True
... | None = False
```

The second is that, in a dependently typed language, subsidiary case-analysis can refine the type of previous arguments. For example:

```
extract-forgotten : {A : Set} {B : A → Set}
   → (f : (x : A) → Maybe (B x))
   → (x : A) → Check (forgetMaybe f x) → (B x)
extract-forgotten f x p  with f x
...                        | Some b = b
extract-forgotten _ _ ()   | None
```

In the second clause, once f x is known to be None, forgetMaybe computes to False, and thus the type of p computes to Void. Thus, we can pattern-match on p with an *absurd pattern* (), which indicates to Agda that the type in question is uninhabited.

Part I

Programming with Logics in Existing Languages

Chapter 3

Security-Typed Programming

The example described in this chapter was developed jointly with Jamie Morgenstern, and published at ICFP 2010 (Morgenstern and Licata, 2010).

Security-typed programming languages allow programmers to specify and enforce security policies, which describe both *access control*—who is permitted to access sensitive resources?—and *information flow*—what are they permitted to do with these resources once they get them? Aura (Jia et al., 2008) and PCML5 (Avijit et al., 2010) enforce access control using dependently typed proof-carrying authorization (PCA): the run-time system requires every access to a sensitive resource be accompanied by a proof of authorization (Appel and Felten, 1999), while the type system aids programmers in constructing correct proofs. Fable (Swamy et al., 2008) and Jif (Chong et al., 2009) enforce information flow properties using type systems that restrict the use of values that depend on private information. Fine (Swamy et al., 2010) combines these techniques to enforce both. These languages' type systems employ a number of advanced techniques, such as dependently typed authorization proofs, indexed monads of computations at a place and on behalf of a principal (Avijit and Harper, 2007), information flow types, and affine types for ephemeral security policies.

In this chapter, we show that security-typed programming can be embedded within a general-purpose dependently typed programming language, Agda. We implement a library, Aglet, which accounts for the major features of existing security-typed programming languages, such as Aura, PCML5, and Fine:

Decentralized Access Control: Access control policies are expressed as propositions in an *authorization logic*, Garg and Pfenning's BL_0 (Garg, 2009b). This permits *decentralized* access control policies, expressed as the aggregate of statements made by different principals about the resources they control. In our embedding, we represent BL_0's propositions and proofs using dependent types, and exploit Agda's type checker to validate the correctness of proofs.

Dependently Typed PCA: Primitives that access resources, such as file system operations, require programmers to provide a proof of authorization, which is guaranteed by the type system to be a well-formed proof of the correct proposition.

Ephemeral and Dynamic Policies: Whether or not one may access a resource is often dependent upon the state of a system. For example, in a conference management server, authors may submit a paper, but only before the submission deadline. Fine accounts for ephemeral policies

using a technique called affine types, which requires a substructural notion of variables. Because Agda does not currently provide substructurality, we show that one can instead account for ephemeral policies using an indexed monad. Following Hoare Type Theory (Nanevski et al., 2008), we define a type $\bigcirc\ \Gamma\ A\ \Gamma'$, which represents a computation that, given precondition Γ, returns a value of type A, with postcondition Γ'. Here, Γ and Γ' are propositions from the authorization logic, describing the state of resources in the system. For example, consider the operation in a conference management server that closes submissions and begins reviewing. We represent this by a computation of type

$\bigcirc$ (InPhase Submission) Unit (InPhase Reviewing)

Given the conference is in phase `Submission`, this computation returns a value of type `Unit`, and the state of the conference has been changed to `Reviewing`. For comparison between the approaches, we adapt Fine's conference management example to our indexed monad. Aglet also permits dynamic acquisition and generation of policies—e.g., generating a policy based on reading the state of the conference management server from a database on startup.

Authentication: Following previous work by Avijit and Harper (2007), we model authentication with an indexed monad of computation on behalf of a principal, which tracks the currently authenticated user. This monad is equipped with a `sudo` operation for switching users, given appropriate credentials. We show that computation on behalf of a principal is a special case of our policy-indexed monad $\bigcirc\ \Gamma\ A\ \Gamma'$.

Spatial distribution: We also show that our policy-indexed monad can be used to model spatial distribution as in PCML5.

Information Flow: Information flow policies constrain the use of values based on what went into computing them, e.g. tainting user input to avoid SQL injection attacks. We represent information flow using well-established techniques, such as indexed monads Russo et al. (2008) and applicative functors (Swamy et al., 2010).

Compile-time and Run-time Theorem Proving: Dependently typed PCA admits a sliding scale between static and dynamic verification. At the static end, one can verify, at compile-time, that a program complies with a statically-given authorization policy. This verification consists of annotating each access to a resource with an authorization proof, whose correctness is ensured by type checking. However, in many programs, the policy is not known at compile time—e.g., the policy may depend upon a system's state. Such programs may dynamically test whether each operation is permitted before performing it, in which case dependently typed PCA ensures that the correct dynamic checks are made and that failure cases are handled. A program may also mix static and dynamic verification: for example, a program may dynamically check that an expected policy is in effect, and then, in the scope of that check, deduce consequences statically. Security-typed languages use theorem provers to reduce the burden of static proofs (as in Fine) and to implement dynamic checks (as in PCML5). We have implemented a certified theorem prover for BL_0, based on a focused sequent calculus. Our theorem prover can be run at compile-time and at run-time, fulfilling both of these roles. The theorem prover also saves programmers from having to understand the details of the authorization logic, as they often do not need to write proofs manually.

In Section 3.1, we show a variety of examples adapted from the literature, which demonstrate that Aglet accounts for programming in the style of Aura, PCML5, and Fine. In Section 3.2, we

$$
\begin{array}{l}
\mathsf{Admin\ says}\ (\forall r.\forall o.\forall f. \\
\quad (\mathsf{HR\ says\ employee}(r) \\
\quad \wedge\ \mathsf{System\ says\ owns}(o, f) \\
\quad \wedge\ o\ \mathsf{says\ mayread}(r, f)) \\
\quad \supset\ \mathsf{mayread}(r, f)) \\
\mathsf{System\ says\ owns}(\mathsf{Jamie}, \mathsf{secret.txt}) \\
\mathsf{HR\ says\ employee}(\mathsf{Dan}) \\
\mathsf{HR\ says\ employee}(\mathsf{Jamie}) \\
\mathsf{Jamie\ says\ mayread}(\mathsf{Dan}, \mathsf{secret.txt}) \\
\mathsf{Jamie\ says\ mayread}(\mathsf{Jamie}, \mathsf{secret.txt})
\end{array}
$$

Figure 3.1: Sample access control policy

describe the implementation of Aglet, including the representation of the logic and the implementation of the theorem prover. We discuss related work on security-typed programming in Section 3.3.

3.1 Examples

In this section, we show that Aglet supports security-typed programming in the style of Aura, PCML5, and Fine by implementing a number of the benchmark examples considered in the literature.

3.1.1 File IO with Access Control

First, we show a dependently typed file system interface, a standard example of security typed programming (Avijit and Harper, 2007; Swamy et al., 2010; Vaughan et al., 2008).

Policy

To begin, we specify an authorization policy for file system operations in BL_0 (Figure 3.1): First, the principal Admin says that for any reader, owner, and file, if human resources says the reader is an employee, and the system administrator says the owner owns the file, and the owner says the reader may read a file, then the reader may read the file. Admin is a distinguished principal whose statements will be used to govern file system operations. Second, the system administrator says Jamie owns secret.txt. Third, human resources says both Dan and Jamie are employees. Fourth, Jamie says Dan and Jamie may read the file. This policy illustrates decentralized access control using the |says| modality: the policy is the aggregate of statements by different principals about resources they control.

For the principal Dan to read secret.txt, it will be sufficient to deduce the goal

$$\mathsf{Admin\ \ says\ \ mayread}(\mathsf{Dan}, \mathsf{secret.txt})$$

This proposition is provable from the above policy because of three properties of says : First, says is closed under instantiation of universal quantifiers (that is, k says $\forall x.A(x)$ entails $\forall x.k$ says $A(x)$). Second, says distributes over implications (k says $(A \supset B)$ entails $((k \text{ says } A) \supset (k \text{ says } B)$). Third, every principal believes that every statement of every other principal has been made (k says A entails k' says $(k \text{ says } A)$)—though it is not the case that every principal believes that every statement of every other principal is *true*. Thus, the goal can be proved by using the first clause of the policy (Admin says ...), instantiating the quantifiers, and using the other statements in the policy to satisfy the preconditions.

In Agda, we represent this first clause as the first element of the following context (list of propositions):

```
Γpolicy =
  (▶ Prin "Admin" says
    (∀e principal · ∀e principal · ∀e filename ·
         let owner = ▷ (iS (iS i0))
             reader = ▷ (iS i0)
             file = ▷ i0 in
       (((▶ Prin "HR" says (a- (Employee · reader)))
         ∧ (▶ Prin "System" says (a- (Owner · (owner, file))))
         ∧ (owner says (a- (Mayread · (reader, file)))))
         ⊃
           (a- (Mayread · (reader, file)))))) ::
  (▶ Prin "Admin" says
    (∀e principal ·
      ∀e filename ·
        (▶ Prin "System" says (a- (Owner · (▷ iS i0, ▷ i0))))
      ⊃
        (a- (MayChown · (▷ iS i0, ▷ i0))))) ::
  []
```

The second element of the list expresses an additional policy clause, not discussed above, which states that an owner of a file may change its ownership. Variables are represented as de Bruijn indices (i0, iS), constants are represented as injections of strings (i.e. Admin is written as ▶ Prin "Admin", where Prin makes a constant out of a string, and ▶ makes a term out of a constant), and atomic propositions are tagged with a *polarity* (a+ or a-), which can be thought of as a hint to the theorem prover. Quantifiers are written ∀e τ · A, where τ is the domain of quantification and A is the body of the quantifier. (Here ∀e is the name of the quantifier, with the e standing for *explicit*—the type τ is given as an explicit argument.) Atomic propositions are written p · t, where p is a proposition constant such as Mayread and t is a term (see Section 3.2.1 for details).

Next, we define a context representing a particular file system state. This context includes all the employee, ownership, and may-read facts mentioned above, with one additional clause saying that Dan may su as Jamie.

```
Γstate =
  (▶ Prin "System" says
    (a- (Owner · (▶ Prin "Jamie", ▶ File "secret·txt"))))
  :: (▶ Prin "HR" says (a- (Employee · (▶ Prin "Dan"))))
  :: (▶ Prin "HR" says (a- (Employee · (▶ Prin "Jamie"))))
  :: (▶ Prin "Jamie" says
    (a- (Mayread · (▶ Prin "Dan", ▶ File "secret·txt"))))
  :: (▶ Prin "Jamie" says
    (a- (Mayread · (▶ Prin "Jamie", ▶ File "secret·txt"))))
  :: (▶ Prin "Admin" says
    (a- (MaySu · (▶ Prin "Dan", ▶ Prin "Jamie"))))
  :: []
Γall = Γpolicy ++ Γstate
```

Finally, we let Γall stand for the append of Γpolicy and Γstate.

Compile-time Theorem Proving

We now explain the use of our theorem prover:

```
goal = a- (Mayread · (▶ Prin "Dan", ▶ File "secret·txt"))
proof? : Maybe (Proof Γall goal)
proof? = prove 15
theProof : Proof Γall goal
theProof = solve proof?
```

The term proof? sets up a call to the theorem prover, attempting to prove mayread(Dan, secret.txt) using the policy specified by Γall. Sequent calculus proofs are represented by an Agda type family $(\Omega; \Delta; \Gamma; \mathsf{k}) \vdash \mathsf{A}$, where Ω binds individual variables, Δ is a context of *claims* assumptions, Γ is context of *truth* assumptions, and k, the *view*, is a principal from whose point of view the judgement is made. Informally, the role of the view is that, in a sequent whose view is k, k says A entails A; see Section 3.2.1 for details about the logic. In this example, Ω and Δ will always be empty, Γ will represent a policy, as above, and the view k will be Prin "Admin"—we abbreviate such a sequent by Proof Γ A. The context and proposition arguments to prove can be inferred by Agda, and so are left as implicit arguments. The term theProof checks that the theorem prover succeeds *at compile-time* in this instance. The function solve has type:

```
solve : ∀ {A} (s : Maybe A) → {p : Check (isSome s)} → A
```

The argument p, of type Check (isSome s), is a proof that s is equal to Some s' for some s'. Because this argument is implicit, Agda will attempt to fill it in by unification, which will succeed when s is definitionally equal to a term of the form Some s'. In this example, the call to the theorem prover in the term proof? proves the goal, computing definitionally to Some s' for a proof s' of mayread(Dan, secret.txt). Thus, we can use solve to extract this proof s'. In general, a call to the theorem prover on a context and a proposition that have no free Agda variables will always be equal to either Some p or to None.

```
◯ : TCtx+ [] → (A : Set) → (A → TCtx+ []) → Set
return : ∀ {Γ A} → A → ◯ Γ A (λ _ → Γ)
_ >>= _ : ∀ {A B Γ Γ' Γ''}
        → (◯ Γ A Γ')
        → ((x : A) → ◯ (Γ' x) B Γ'')
        → ◯ Γ B Γ''
weakenPre : ∀ {A Γ Γ' Γn}
  → (Good Γn → Good Γ)
  → ◯ Γ A Γ' → Γ ⊆ Γn → ◯ Γn A Γ'
weakenPost : ∀ {A Γ Γ' Γn}
  → ◯ Γ A Γ'
  → ((x : A) → (Γn x ⊆ Γ' x))
  → ((x : A) → (Good (Γ' x) → Good (Γn x)))
  → ◯ Γ A Γn
getLine : ∀ {Γ} → ◯ Γ String (λ _ → Γ)
print : ∀ {Γ} → String → ◯ Γ Unit (λ _ → Γ)
error : ∀ {A Γ Γ'} → String → ◯ Γ A Γ'
acquire : ∀ {A Γ Γ'} → (Γn : TCtx+ [])
        → (Good Γ → Good (Γn ++ Γ))
        → ◯ (Γn ++ Γ) A Γ' → ◯ Γ A Γ'
        → ◯ Γ A Γ'
```

Figure 3.2: Monadic IO with Authorization

Computations

We present a monadic interface for file operations in Figures 3.3 and 3.2. This figure shows both the generic IO operations, as well as three file-specific operations for reading, creating, and changing the owner of a file. The type ◯ Γ A Γ' represents a computation with precondition Γ and postcondition Γ'. The Agda type of a context is TCtx+ [] (a context of positive truth assumptions, with no free individual variables—see Section 3.2.1). The postcondition is a function from A's to contexts, so the postcondition may depend on the computation's result (see create below). The generic operations are typed as follows: Because return is not effectful, its postcondition is its precondition. Bind (>>=) chains together two computations, where the postcondition of the first is the precondition of the second. Both pre- and postconditions can be weakened to larger and smaller contexts, respectively; the Good predicate can be ignored until Section 3.1.1 below. Primitives like getLine (reading a line of input) and print do not change the state and do not require proofs. The postcondition of error is arbitrary, as it never terminates successfully. The remaining computations are defined as follows:

```
sudo : ∀ {Γ A Γ' Δ Δ'} → (k1 k2 : _)
  → Replace (a+ (As · k1)) (a+ (As · k2)) Γ Δ
  → ((x : A) → Replace (a+ (As · k2)) (a+ (As · k1))
    (Δ' x) (Γ' x))
  → (Proof Γ (a- (MaySu · (k1, k2))))
  → ◯ Δ A Δ'
  → ◯ Γ A Γ'
read : ∀ {Γ} (k : _) (file : _)
  → Proof Γ ((a- (Mayread · (k, file)))
    ∧ (a+ (As · k)))
  → ◯ Γ String (λ _ → Γ)
create : ∀ {Γ} (k : _)
  → Proof Γ ( (a- (User · k))
    ∧ (a+ (As · k)))
  → ◯ Γ String
    (λ new → (▶ Prin "System" says
      (a- (Owner · (k, ▶ File new)))) :: Γ)
chown : ∀ {Γ Δ} → (k k1 k2 : _) → (f : _)
  → Replace (▶ Prin "System" says (a- (Owner · (k1, f))))
    (▶ Prin "System" says (a- (Owner · (k2, f))))
    Γ Δ
  → (Proof Γ ((a+ (As · k))
    ∧ (a- (MayChown · (k, f)))))
  → ◯ Γ Unit (λ _ → Δ)
```

Figure 3.3: Monadic File IO with Authorization

Read The function read takes a principal k, a file f, and a proof argument. The proof ensures that the principal k is authorized to access the file (Mayread (k, f)) and that the principal k is the currently authenticated user (As (k)). An alternate type for read would put these facts in the context Γ, rather than as a separate proof argument; we do not take this approach because, as discussed below, Γ will contain only atomic facts that are known about the policy, and not arbitrary logical consequences of them. We use the proposition As to model computation on behalf of a principal (Avijit and Harper, 2007). The proof is checked in the context Γ that is the precondition of the computation, ensuring that it is valid in the current state of the world. read delivers the contents of the file and leaves the state unchanged.

An example call to read looks like this:

```
Γj = Γall as "Jamie"


jread : ◯ Γj String (λ _ → Γj)
jread = read (▶ Prin "Jamie") (▶ File "secret·txt")
  (solve (prove 17))
jreadprint : ◯ Γj Unit (λ _ → Γj)
jreadprint = jread ≫= λ x →
  print ("the secret is: " ↑ x)
```

The function call Γall as k is shorthand for adding the proposition As (k) to the context Γall. The computation jread reads the file secret.txt as principal Jamie; the proof argument is supplied by a call to the theorem prover, which statically verifies that the required fact is derivable from the policy given by Γall. The computation jreadprint reads the file and then prints the result.

Create The type of create is similar to read, in that it takes a principal and a proof that the principal can create a file (in this case, the fact that the principal is a registered user is deemed sufficient). It returns a String, the name of the created file, and illustrates why postconditions must be allowed to depend on the return value of the computation: the postcondition says that the principal is the owner of the newly created file. Thus, after a call to create (k), the postconditions signify System says Owner (k, f), where f is the name of the new file.

Chown To specify chown, we use a type Replace x y Γ Δ, which means that Δ is the result of replacing exactly one occurrence of x in Γ with y. Replace (whose definition is not shown) is defined by saying that (1) there is a de Bruijn index i showing that x is in Γ and (2) Δ is equal to the output of the function replace y i, which recurs on the index i and replaces the indicated element by y. The type of chown should be read as follows: if the principal k as whom the computation is running has the authority to change the owner of a file, and replacing owns (k, f) with owns (k', f) in Γ produces Δ, then we can produce a computation which changes the owner of f from k to k', leaving the remaining context unchanged.

Next, we show an example call to chown, using a context Γstate' that is the result of replacing the fact that Jamie owns secret.txt with Dan owning that file. The computation dchown runs as Dan; it changes the owner of the file from Dan to Jamie, and then runs a computation drdprnt,

defined below, that reads the file. proveReplace is a tactic used to prove that Γall' is Γall with the ownership of secret.txt changed. solve (prove 15) calls the theorem prover to statically verify that Dan has permission to chown secret.txt.

```
Γstate' = replace {_} {Γstate}
  (▶ Prin "System" says
    (a- (Owner · (▶ Prin "Dan", ▶ File "secret·txt"))))
  i0
Γall' = Γpolicy ++ Γstate'
dchown : ◯ (Γall' as "Dan") Unit (λ _ → Γall as "Dan")
dchown = chown (▶ Prin "Dan") (▶ Prin "Dan") (▶ Prin "Jamie")
  (▶ File "secret·txt")
  (solve proveReplace) (solve (prove 15))
  ≫ drdprnt
```

Sudo Following Avijit and Harper (2007), we now give a well-typed version of the Unix command sudo, which allows switching principals during execution. A first cut for the type of sudo is as follows:

```
sudo1 : ∀ {Γ A Γ'} → (k1 k2 : _)
→ (Proof Γ (a- (MaySu · (k1, k2))))
→ ◯ ((a+ (As · k2)) :: Γ) A (λ _ → (a+ (As · k2)) :: Γ')
→ ◯ ((a+ (As · k1)) :: Γ) A (λ _ → (a+ (As · k1)) :: Γ')
```

If there is a proof that k1 may sudo as k2 (e.g., a password was provided), and As (k1) is in the precondition, then it is permissible to run a subcomputation as k2. This subcomputation has a postcondition saying that it terminates running as k2, and then the overall computation returns to running as k1. Because our contexts are ordered (represented as lists rather than sets), sudo has the type in Figure 3.3, which allows the As facts to occur anywhere in the context. sudo's type may be read: if replacing As (k1) with As (k2) in Γ equals Δ, and if replacing As (k2) with As (k1) in Δ' equals Γ', and k2 has permission to su as k1, then a computation with preconditions Δ and postconditions Δ' can produce a computation with preconditions Γ and postconditions Γ'.

The following example call to sudo defines a computation as Dan that su's as Jamie to run the computation jreadprint defined above:

```
drdprnt : ◯ (Γall as "Dan") Unit (λ _ → Γall as "Dan")
drdprnt = sudo (▶ Prin "Dan") (▶ Prin "Jamie")
  (solve proveReplace)
  (λ _ → solve proveReplace)
  (solve (prove 15))
  jreadprint
```

This requires proving that Γstate as "Jamie" and Γstate as "Dan" are related by replacing As (Prin "Jamie") with As (Prin "Dan") (in both directions). Our tactic proveReplace proves all of these equalities. Additionally, the theorem prover statically verifies Dan may su as Jamie under the policy Γall as "Dan".

Acquire The function acquire allows a program to check whether a proposition is true in the state of the world. This construct is inspired by acquire in PCML5, but there are slight differences: in PCML5, acquire does theorem proving to prove an arbitrary proposition from the policy, whereas here acquire only verifies the truth of state-dependent atomic facts (which have no evidence) and statements of principals (whose only evidence is a digital signature (Avijit et al., 2010; Jia et al., 2008)). The function acquire takes two continuations: one to run if the check is successful, whose precondition is extended with the proposition, and an error handler, whose precondition is the current context, to run if the check fails. In fact, we allow acquire to test an entire context at once: given a context Γn, a computation with preconditions Γ extended with Γn (the success continuation), and a computation with preconditions Γ (the error continuation), acquire returns a computation with preconditions Γ. We use the notation acquire Γn / _ no⇒ s yes⇒ f to write a call to acquire in a pattern-matching style. The _ elides a Good argument, which is explained below.

```
main : ◯ [] Unit (λ _ → [])
main = acquire (Γall as "Jamie") / _
  no⇒ error "acquiring policy failed"
  yes⇒ weakenPost jreadprint (λ _ ()) _
```

This example call begins and ends in the empty context. The call to acquire examines the system state to check the truth of each of the propositions in Γall as "Jamie". If all of these are true, then we run jreadprint and use weakening to forget the postconditions. If some proposition cannot be verified, then main calls error.

Verifying Policy Invariants

When authoring the above monadic signature for file IO, the programmer may have in mind some invariants to which policies Γ must adhere. For example, a call to chown (above) would have unexpected consequences if there ever were more than one copy of System says owns (k, f) in Γ (only one copy would be replaced, leaving a file with two owners in the postcondition). Our interface permits programmers to specify context invariants using a predicate Good Γ. The intended invariant of our interface is that a monadic computation ◯ Γ A Γ' should have the property that Γ' satisfies Good if Γ does. To achieve this, the weakening operations and acquire require the preconditions Γ be accompanied by a proof of Good Γ, and the programmer must verify that operations such as read, chown, and sudo preserve the invariant. Because of this invariant, it is not necessary to make each monadic operation require a proof that the precondition is Good. This means, that when writing a client program, the programmer needs only to verify that the initial policy and those in calls to weakening and acquire satisfy the invariants.

In the above examples, we took Good to be the trivially true invariant, so the proofs could be elided with an _. As mentioned above, a useful invariant to enforce is that for every file f there is at most one statement of the form System says Owner (_, f) in the context. This is defined in Agda as follows:

```
Good : TCtx+ [] → Set
Good Γ = ∀ {k k' f}
→ (a : (▶ Prin "System" says (a- (Owner · (k, f)))) ∈ Γ)
→ (b : (▶ Prin "System" says (a- (Owner · (k', f)))) ∈ Γ)
→ Equal a b
```

Then we may prove that the postcondition of each operation is Good if the precondition is; e.g.

```
ChownPreservesGood : ∀ {Γ Δ k1 k2 f}
→ Replace (▶ Prin "System" says (a- (Owner · (k1, f))))
  (▶ Prin "System" says (a- (Owner · (k2, f))))
  Γ Δ
→ Good Γ → Good Δ
```

In the companion code, we revise the above examples so that they maintain this invariant, using a tactic to generate the proofs.

3.1.2 File IO with Access Control and Information Flow

Next, we extend the above file signature with information flow, adapting an example from Fine (Swamy et al., 2010). First, we define a type Tracked A L which represents a value of type A tracked with security level L, where L is a list of filenames and ⊔ appends two lists. Following Fine, we define Tracked as an abstract functor that distributes over functions (though different type structures for information flow, such as an indexed monad (Russo et al., 2008), can be used in other examples):

```
Tracked : Set → Label → Set
fmap : ∀ {A B L} → (A → B) → Tracked A L → Tracked B L
_⊙_ : ∀ {A B L1 L2} → Tracked (A → B) L1
  → Tracked A L2 → Tracked B (L1 ⊔ L2)
```

An application f ⊙ x joins the security levels of the function and the argument.

Next, we give flow-sensitive types to read and write: read tags the value with the file it was read from, and write requires a proof of MayAllFlow provs file, representing the fact that all of the files upon which the written string depends may flow into file.

```
read : ∀ {Γ} (k : _) (file : _)
→ Proof Γ ((a- (Mayread · (k, file))) ∧ (a+ (As · k)))
→ ◯ Γ (Tracked String [file]) (λ _ → Γ)

write : ∀ {Γ provs} (k : _) (file : _)
→ Tracked String provs
→ Proof Γ ((a- (Maywrite · (k, file)))
  ∧ (a+ (As · k))
  ∧ (MayAllFlow provs file))
→ ◯ Γ Unit (λ _ → Γ)
```

For example, we can read two files and write their concatenation to secret.txt:

```
go : ◯ (Γ as "Jamie") Unit (λ _ → (Γ as "Jamie"))
go = read (▶ Prin "Jamie") (▶ File "file1.txt")
       (solve (prove 15)) ≫= λ s →
     read (▶ Prin "Jamie") (▶ File "file2·txt")
       (solve (prove 15)) ≫= λ s' →
     write (▶ Prin "Jamie") (▶ File "secret·txt")
       ((fmap String.string-append s) ⊙ s')
       (solve (prove 15))
```

Here the theorem prover shows that both file1.txt and file2.txt may flow into secret.txt, according to the policy. This proof obligation results from the fact that
(fmap String.string-append s) ⊙ s'
has type
Tracked String ["file1.txt", "file2.txt"].

3.1.3 Spatial Distribution with Information Flow

PCML5 investigates PCA for the spatially distributed programming language ML5 (Murphy VII, 2008). Here, we show how to embed an ML5-style type system, which can be combined with the above techniques for access control and information flow. PCML5 considers additional aspects of distributed authorization, such as treating the policy itself as a distributed resource, which we leave to future work.

ML5 tracks where resources and computations are located using modal types of the form A @ w. For example, database.read : (key → value) @ server says that a function that reads from the database must be run at the server, while javascript.alert : (string → unit) @ client says that a computation that pops up a browser alert box must be run at the client. Network communication is expressed in ML5 using an operation get : (unit → A) @ w → A @ w' that (under some conditions which we elide here) goes to w to run the given computation and brings the resulting value back to w'. In other work (Licata and Harper, 2010), we have shown how to build an ML5-like type system on top of an indexed monad of computations at a place, ◯ w A, with a rule get : ◯ w' A → ◯ w A. Here, observe that this monad indexing can be represented using a proposition At (w), where get is given a type analogous to sudo:

```
get : (w1 w2 : _) → ∀ {Γ A Γ' Δ Δ'}
  → Replace (a+ (At · w1)) (a+ (At · w2)) Γ Δ
  → Replace (a+ (At · w2)) (a+ (At · w1)) Δ' Γ'
  → ◯ Δ A (λ _ → Δ')
  → ◯ Γ (Tracked A w2) (λ _ → Γ')
```

Additionally, we combine spatial distribution with information flow, tagging the return value of the computation with the world it is from. The postcondition must be independent of the return value, as there is in general no coercion either way between A and Tracked A L.

Information flow can be used in this setting to force strings to be escaped before they are sent back to the client—e.g. to prevent SQL injection attacks:

$$\mathsf{sanitize} \;:\; \mathsf{Tracked\ String}\ (\blacktriangleright\ \mathsf{client}) \to \mathsf{HTML}$$
$$\mathsf{str} \;:\; \mathsf{Tracked\ String}\ (\blacktriangleright\ \mathsf{server}) \to \mathsf{HTML}$$

Strings from the client must be escaped before they can be included in an HTML document, whereas strings from the server are assumed to be non-malicious, and can be included directly.

3.1.4 ConfRM: A Conference Management System

Swamy et al. (2010) present an example of a conference management server, ConfRM, adapted from CONTINUE (Krishnamurthi, 2003) and its access control policy (Dougherty et al., 2006). Here, we show an excerpt of an authorization policy for ConfRM, a proof-carrying monadic interface to the computations which perform actions, and the main event loop of the server. This example uses ephemeral policies: authorization to perform actions, such as submitting a paper or a review, depend on the phase of the conference (submission, notification,...).

Policy

We formalize ConfRM's policy using terms of various types: actions represent requests to the web server; principals represent users; papers and strings are used to specify actions; roles define whether a user is an Author, PCMember, and so on. The policy is also dependent on the phase of the conference (e.g., an Author may submit a paper during the submission phase). The proposition $\mathsf{May} \cdot (\mathsf{k}, \mathsf{a})$ states that k may perform action a. Each action is a first-order term constructed from some arguments (e.g., Submit, Review, Readscore, Read all have papers, while Progress has two phases, the phase the conference is in before and after it is progressed).

Fine specifies the policy as a collection of Horn clauses, which are simple to translate to our logic, as in the following clause:

```
clause1 =
 ( (∀e principal · ∀e string ·
    let
      author = ▷ iS i0
      papername = ▷ i0
    in
     (( (a- (InPhase · (▶ Submission))) ∧
       ( (a- ( InRole · (author  , ▶ Author)))))
      ⊃ (a- (May · (author , (Submit · papername)))))))
```

This proposition reads: for all authors and paper names, if the conference is in the submission phase, and the principal is an author, then the principal may submit a paper. We have also begun to reformulate the policy using the `says` modality, e.g. to allow authors to share their paper scores with their coauthors.

```
saysClause =
 ( (∀e principal · ∀e paper · ∀e principal ·
   let primary = ▷ i0
       paper = ▷ (iS i0)
       coauthor = ▷ (iS (iS i0)) in
   (( ( (a- (InPhase · (▶ Notification)))) ∧
      ( (a- (Author · (primary , paper) ))) ∧
      (primary says (a- (May · (coauthor ,
                                (Readscore · paper))))))
    ⊃ (a- (May · (coauthor , (Readscore · paper)))))))
```

This rule states that, for any principal `author`, paper `paper`, and principal `coauthor`, if the conference is in notification phase, and `author` is the author of `paper`, and `author` says `coauthor` may read the scores for `paper`, then `coauthor` may read the scores for `paper`. Similarly, using `says`, it is straightforward to specify a policy allowing PC members to delegate reviewing assignments to subreviewers.

Actions

Rather than defining a command for each action—doRead, doSubmit, etc.— we use type-level computation to write one command for processing all actions; this simplifies the code for the main loop presented below and allows for straightforward addition of actions. The generic command for processing an action, doaction, has the following type:

```
doaction : ∀ {Γ} (k : _) (a : _) → (e : ExtraArgs Γ a)
→ Proof Γ (a- (May · (k, a))) ∧ (a+ (As · k))
→ ◯ Γ (Result a) (λ r → PostCondition a Γ e k r)
```

doaction takes a principal k, an action a to perform, and some ExtraArgs for a, along with a proof that the computation is running as k, and that k may perform a. In this example, a Proof abbreviates a sequent whose view is PCChair, rather than Admin. It returns a Result, and has a PostCondition, both of which are dependent upon the action being performed. In Agda, ExtraArgs, Result, and PostConditions are functions defined by recursion on actions, which compute a Set, a Set, and a context, respectively.

Several actions, such as Submitting a paper, require extra data that is not part of the logical specification (e.g., the contents of the paper should not be part of the proposition which authorizes it to be submitted). ExtraArgs produces the set of additional arguments each action requires.

```
ExtraArgs : TCtx+ [] → Term [] (▶ action) → Set
ExtraArgs Γ (Review · _) = Term [] (▶ string)
ExtraArgs Γ (Submit · _) = Term [] (▶ string)
ExtraArgs Γ (Progress · (p1, p2)) =
  Σ (λ Δ → Replace (a- (InPhase · p1)) (a- (InPhase · p2)) Γ Δ)
ExtraArgs Γ _ = Unit
```

Reviews and paper submissions require their contents, represented as terms of type string (the Agda type Term [] (▶ string) is an injection of strings into the language of first-order terms that

we use to represent propositions, as described in Section 3.2 below). Progressing the phase of the conference requires a proof that the conference is in the first phase, along with a new context in the resulting phase, which we represent by a pair of a new context Δ and a proof of Replace.

Next, we specify the result type of an action:

```
Result : Term [] (▶ action) → Set
Result (Submit · _) = Term [] (▶ paper)
Result (Review · _) = Unit
Result (BeAssigned · _) = Unit
Result (Readscore · _) = String
Result (Read · _) = String
Result (Progress · _) = Unit
```

Readscore and Read return a paper's reviews and contents, while submit produces a Term [] (▶ paper), a unique id for the paper.

Finally, we define the PostCondition of each action, which is dependent upon the action itself, the precondition, the extra arguments for the action, the principal performing the action, and the Result of the action. Submitting a paper extends the preconditions with two propositions: one saying the paper has been submitted, and one saying the submitting principal is its author. Reviewing and Assigning a paper add that the paper is reviewed by or assigned to the principal, respectively. Readscore and Read leave the conditions unchanged. The postcondition of Progress is the first component of its ExtraArgs, i.e. the context determined by replacing the current phase with the resulting one.

```
PostCondition : (a : Term [] (▶ action)) (Γ : TCtx+ [])
  → ExtraArgs Γ a → (k : Term [] (▶ principal))
  → Result a → TCtx+ []
PostCondition (Submit · y)      Γ e k r = (a- (Submitted · r)) :: (a- (Author · (k, r))) :: Γ
PostCondition (Review · y)      Γ e k r = (a- (Reviewed · (k, y))) :: Γ
PostCondition (BeAssigned · y) Γ e k r = (a- (Assigned · (k, y))) :: Γ
PostCondition (Readscore · y)  Γ e k r = Γ
PostCondition (Read · y)        Γ e k r = Γ
PostCondition (Progress · (ph1, ph2)) Γ e k r = (fst e)
```

In writing the main server loop, we will use the following monadic wrapper of our theorem prover, in order to test at run time whether a given proposition holds in the current state of the server:

```
prove/dyn : ∀ {Γ1} → Nat → (Γ : TCtx+ []) →
  (A : Propo- []) →
  ◯ Γ1 (Maybe (Proof Γ A)) (λ _ → Γ1)
```

Server Main Loop

In Figure 3.4 we show the code for the main loop of the ConfRM server, implemented using the interface described above. The main loop serves requests made by principals who wish

```
fix : ∀ {A Γ'}
  → ((∀ {Γ} → ◯ Γ A Γ') → (∀ {Γ} → ◯ Γ A Γ'))
  → (∀ {Γ} → ◯ Γ A Γ')

main : ∀ {Γ} → ◯ Γ Unit (λ _ → [])
main = fix loop where
  loop : (∀ {Γ} → ◯ Γ Unit (λ _ → [])) →
           (∀ {Γ} → ◯ Γ Unit (λ _ → []))
  loop rec {Γ} =
    prompt "Enter an action:" >>= λ astr →                    {-1 -}
      case (parseAction astr)
        None⇒ error "Unknown action"
        Some⇒ λ actionArgs →
          let a = (fst actionArgs)
                  args = (snd actionArgs) in
            prompt "Who are you?" >>= λ ustring →             {-2 -}
              let u = parsePrin ustring in
                acquire [( (a- (MaySu · (▶ Prin "Admin", u))))]   {-3 -}
                  / _
                  no⇒ error "Unable to su"
                  yes⇒ case make-replace  {-4 -}
                    None⇒ error "oops, not running as admin"
                    Some⇒ λ asadmin →
                      case (inputToE a _ args)                {-5 -}
                        None⇒ error "Bad input (e.g. not in phase)"
                        Some⇒ λ args →
                          (sudo (▶ Prin "Admin") u            {-6 -}
                            (snd asadmin)
                            (\x → (snd (repAsPost (snd asadmin)
                              {a} x)))
                            (lfoc i0 init-)
                            (prove/dyn 15 _ _ >>=             {-7 -}
                              none⇒ error "Unauthorized action"
                              some⇒ λ canDoAction →
                                doaction u a args canDoAction))  {-8 -}
                          >>= λ _ → rec                       {-9 -}
```

Figure 3.4: ConfRM Main Loop

to perform actions. Because the requests are not determined until run-time, and authorization depends on the system state (the phase of the conference, the role of a principal), this example uses entirely dynamic verification of security policies: the server dynamically checks that each request is authorized just before performing it, using our theorem prover at run-time. The type system ensures that the appropriate dynamic check is made. Informally, the server loop works by (1) reading in an action and its arguments, (2) reading in a principal, (3) acquiring the credentials to su as that principal, (4) computing the precondition of the su, (5) computing the postconditions of performing the action, (6) su-ing as the principal, (7) proving the principal may perform the action, (8) performing the action, and (9) recurring. The fact that we have coalesced all of the actions into one primitive command makes this code much more concise than it would be otherwise, when we would have to repeat essentially this code as many times as there are actions.

This code is rendered in Agda as follows. fix permits an IO computation to be defined by general recursion. Because its type is restricted to the monad, it does not permit non-terminating elements of other types, such as Proof. This fixed-point combinator abstracts over the precondition, so it may vary in recursive calls, but leaves the postcondition fixed throughout the loop; we leave more general loop invariants to future work. First, main is given the type $\forall\ \lambda\ \Gamma \rightarrow \bigcirc\ \Gamma\ \mathsf{Unit}\ (\lambda\ \_ \rightarrow [\,])$: given any precondition, the computation returns unit and an empty postcondition (we do not expect to run any code following main so it is not worthwhile to track the postconditions). main is defined by taking the fixed point of the axillary function loop, which is abstracted over the recursive call. On line (1), the loop prompts the user to enter an action to perform, parseAction then parses the string to produce a : action and args: InputArgs, and raises an error otherwise. (2) The loop prompts for a username, parses it into a Term [] principal. (3) The loop attempts to acquire credentials that `"Admin"` may su as the principal (e.g., by prompting for a password). (4) The loop calls the functions make-replace to produce the preconditions for the su, by replacing (As (Prin `"Admin"`)) with a+ (As u). (5) The loop calls inputToE to produce the ExtraArgs for the action from the args; for Progress, this function computes the postcondition of the action from the current context. (6) The loop su-s as the principal. The first replace argument to su is the result of step (4), the proof argument is the assumption acquired in step (3), the second replace argument is discussed below. (7) The loop calls the theorem prover at run-time to prove the principal may perform the requested action. (8) The loop calls doaction and (9) recurs.

The second replace argument to su is generated using a proof that As is preserved in the PostCondition of an action:

```
postPreservesAs : ∀ {a Γ e k r k'}
  → (a+ (As · k') ∈ Γ)
  → ((a+ (As · k')) ∈ PostCondition a Γ e k r)
```

This is another example of using Agda to verify invariants of the pre- and post-conditions, as in Section 3.1.1.

Dynamic Policy Acquisition

Finally, we describe an example of dynamic policy acquisition in Figure 3.5: we read the reviewers' paper assignments from a database, parse the result into a context, acquire the context,

```
getReviewerAsgn : ∀ {Γ} → String →
  ◯ Γ (List (List String)) (λ _ → Γ)
parseReviewers : List String → TCtx+ []


mkPolicy : ∀ {Γ} → ◯ Γ (TCtx+ []) (λ _ → Γ)
mkPolicy = getReviewerAsgn "papers.db" >>= λ asgn →
  return (ListM.fold [] (λ x → λ y →
    parseReviewers x ++ y) asgn)
start = mkPolicy {[]} >>= λ ctx →
  acquire ctx / _
    no⇒ error "policy not accepted"
    yes⇒ main
```

Figure 3.5: ConfRM Policy Acquisition

and start the main server loop with those preconditions. This is simple in a dependently typed language because contexts themselves are data. The function getReviewerAsgn takes a string, representing a path to the database, and returns the list of reviewers for each paper. The function parseReviewers then turns each of these lists into lists of propositions, each stating the parsed reviewer is a reviewer of the paper. A more realistic ConfRM implementation would read a variety of other propositions from the database as well (which papers have been submitted, reviewed, etc.) The computation mkPolicy calls getReviewerAsgn and parses the results. The computation start uses mkPolicy to generate an initial policy, acquires these preconditions, and starts the main sever loop.

3.2 Implementation

Our Agda implementation consists of about 1400 lines of code. We have also written about 1800 lines of example code in the embedded language, including policies, monadic interfaces to primitives, and example programs. In this section, we describe the implementation of the logic, the theorem prover, and the indexed monad.

3.2.1 Representing BL_0

BL_0 (Garg, 2009b) extends first-order intuitionistic logic with the modality k says A. While a variety of definitions of says have been studied (Abadi (2008) overviews some of the approaches), in BL_0, says is treated as a necessitation ($\Box$) modality, and *not* as a lax modality (i.e. a monad) (Abadi, 2006; Avijit and Harper, 2007; Garg and Pfenning, 2006; Jia et al., 2008). The definition of says in BL_0 supports *exclusive delegation*, where a principal delegates respon-

sibility for a proposition to another principal, without retaining the ability to assert that proposition himself. For example, consider a policy that payroll says $\forall t.(\mathsf{HR}\ \mathsf{says}\ \mathsf{employee}(t)) \supset \mathsf{MayBePaid}(t)$. Under what circumstances can we conclude payroll says MayBePaid(Alice)? The fact that HR says employee(Alice) should be sufficient. However, the fact that payroll says employee(Alice) should not, as the intention of the policy is that payroll delegates responsibility for the employee predicate to human resources, without retaining the ability to assert employee instances itself. When says is treated as a lax modality, payroll says employee(Alice) implies payroll says HR says employee(Alice), which is enough to conclude the goal. Abstractly, we wish k says A to imply k' says $(k$ says $A)$, but not k says $(k'$ says $A)$. The modality satisfies several other axioms; in a Hilbert-style combinator system, it can be described as follows:

$$\frac{\vdash s}{\vdash k\ \mathsf{says}\ A} \qquad \text{(Necessity)}$$

$k\ \mathsf{says}\ (A \supset B) \supset (k\ \mathsf{says}\ A \supset k\ \mathsf{says}\ B)$ (K)
$(k\ \mathsf{says}\ A) \supset (k'\ \mathsf{says}\ (k\ \mathsf{says}\ A))$ (I)
$k\ \mathsf{says}\ ((k\ \mathsf{says}\ A) \supset A)$ (C)
$(k\ \mathsf{says}\ A) \supset (k'\ \mathsf{says}\ A)$ if $k' \geq k$ (S)

The last axiom refers to a *delegation order* on principals, which automatically includes the statements of one principal into another.

Terms, Types, and Atomic Propositions

In the above examples, we used a variety of atomic propositions (Mayread, Owns, etc.), which refer to several datatypes (principals, papers, conference phases, etc.). Though we are attempting to do this example in "raw Agda,", we could not help but embed a simple logical framework, so that the representation of BL_0 and its theorem prover can be parametrized over such datatypes and atomic propositions. In particular, we implement the simplest possible logical framework: first-order terms, with free variables, over a given signature. This allows us to specify the types, terms, and propositions for an example concisely, while exploiting a datatype-generic definition of the structural properties when we state the inference rules of the logic. The following excerpt from the signature for ConfRM illustrates what one writes to instantiate the logical framework to a specific example:

```
data BaseType : Set where
  string paper role action phase principal : BaseType

data Const : BaseType -> Set where
  Prin  : String -> Const principal
  Paper : String -> Const paper
  PCChair Reviewer Author Public : Const role
  Init Presubmission Submission ... : Const phase

data Func : BaseType -> Type -> Set where
  Review BeAssigned ...  : Func action (▶ paper)
  Progress  : Func action  (▶ phase ⊗ ▶ phase)
```

```
data Atom : Type -> Set where
  InPhase : Atom (▶ phase)
  Assigned ... : Atom (▶ principal ⊗ ▶ paper)
  May :  Atom (▶ principal ⊗ ▶ action)
  As : Atom (▶ principal)
```

The programmer defines a datatype of base types, a datatype giving constants of each type, a datatype of function symbols, and a datatype of atomic propositions over a given type. Additionally, the programmer must define a couple of operations on these types (equality, enumeration of all elements of a finite type) which in a future version of Agda could be generated automatically (Altenkirch and McBride, 2003).

My First Logical Framework

First, we define an interface that formalizes the notion of a *signature*: what the programmer specifies to instantiate the framework to a particular example. The above datatypes BaseType, Const, etc. will constitute one such signature. We represent this in Agda with a record type:

```
record TermSig : Set1 where
  field
    BaseType : Set
    Const : BaseType → Set
    typeEq : (A B : BaseType) → Maybe (Id A B)
    constEq : ∀ {A : BaseType} → (C C' : Const A) → Maybe (Id C C')
    allConst : (extraStrings : List String) → {A : BaseType} → List (Const A)
  module Typ = Types BaseType
  field
    Func : BaseType → Typ.Type → Set
    allFuncs : (τ : BaseType) → List (Σ (λ (A : Typ.Type) → Func τ A))
    funcEq : ∀ {τ1 τ2 : BaseType} {A1 A2 : Typ.Type}
       → (C1 : Func τ1 A1) → (C2 : Func τ2 A2)
       → Maybe (Id τ1 τ2 × Id A1 A2 × HId C1 C2)
```

The programmer defines a set of BaseTypes, and a set Const of constants of each BaseType. When we implement the theorem prover below, we will need to compare types and terms for equality, and enumerate all terms of a given signature, so we additionally require functions that compare BaseTypes and Constants and enumerate all Constants.

The parametrized module Types defines the types over a given collection of base types:

```
module Types (BaseType : Set) where
  data Type : Set where
    ▶_ : BaseType → Type
    _⊗_ : Type → Type → Type
    unit : Type
```

Types are BaseTypes, unit, and pair types $(\tau 1 \otimes \tau 2)$.

Returning to TermSig, the programmer may additionally specify a collection of function symbols Func τ A, which require an argument of type A and produce a term of type τ, along with enumeration and equality for function symbols.

Given an instance of this signature the framework provides a datatype of terms and a collection of operations on it. First, the datatype of terms:

```
ICtx = List BaseType
data Term : ICtx → Type → Set where
  ▷_ : ∀ {Ω τ} → τ ∈ Ω → Term Ω (▶ τ)
  ▶_ : ∀ {Ω τ} → Const τ → Term Ω (▶ τ)
  _,_ : ∀ {Ω τ1 τ2} → Term Ω τ1 → Term Ω τ2 → Term Ω (τ1 ⊗ τ2)
  [] : ∀ {Ω} → Term Ω unit
  _·_ : ∀ {Ω τ A} → Func τ A → Term Ω A → Term Ω (▶ τ)
```

The terms over a signature are given by a datatype Term Ω τ, where Ω, an individual context (ICtx), represents the free variables of the term. An ICtx is a list of BaseTypes, and represents a context of individual variables—e.g. the context $x_1 : \tau_1, \ldots, x_n : \tau_n$ will be represented by the list $\tau_1 :: ... :: \tau_n :: []$. Variables are represented by well-scoped de Bruijn indices, which are pointers into such a list—i0 says $x \in (x :: l)$, and iS says that $x \in (y :: l)$ if $x \in l$. Terms are either variables $(\triangleright i)$, where $i : \tau \in \Omega$ is a de Bruijn index, constants, applications of function symbols $(f \cdot t)$, or [] and $(t1, t2)$ for unit and product types.

Next, we equip this datatype of terms with a variety of operations, generally in the signature. First, we write $\Omega \subseteq \Omega'$ for $\forall \{\tau\} \to \tau \in \Omega \to \tau \in \Omega'$ and define weakening (as well as exchange and contraction):

```
weakenTerm : ∀ {Ω Ω' τ} → Term Ω τ → Ω ⊆ Ω' → Term Ω' τ
```

Similarly, we can define substitution:

```
substTerm : ∀ {Ω τ Ω'}
  → Term Ω τ → (∀ {τ} → τ ∈ Ω → Term Ω' (▶ τ))
  → Term Ω' τ
```

Next, we specialize substitution to the last variable, using the identity substitution on Ω:

```
substTermLast : ∀ {Ω τ τ'} → Term (τ :: Ω) τ' → Term Ω (▶ τ) → Term Ω τ'
```

Finally, we define equality and enumeration:

```
varEq : {Ω : ICtx} {τ : BaseType} → (x y : τ ∈ Ω) → Maybe (Id x y)
termEq : ∀ {Ω τ} → (T1 T2 : Term Ω τ) → Maybe (Id T1 T2)
allVars : (Ω : ICtx) {τ : _} → List (τ ∈ Ω)
allTermsGen : List String → (Ω : _) (A : _) → List (Term Ω A)
```

allTermsGen only terminates if the signature is non-recursive; we could build a proof of this fact into TermSig, but for now we have taken the shortcut of turning off the termination checker.

Next, we prove some properties of these operations generically. First, weakening by the identity is the identity:

```
IsId : ∀ {Ω} → (Ω ⊆ Ω) → Set
IsId {Ω} w = {τ : BaseType} (i : τ ∈ Ω) → Id i (w i)
weakenTerm-id : ∀ {Ω} → {w : Ω ⊆ Ω} → IsId w
  → {τ : Type} (t : Term Ω τ) → Id (weakenTerm t w) A
```

Because Agda equality is intensional, it is too strong to demand Id w (λ x → x); thus, we use IsId, which says that is the identity on all arguments.

Second, we prove that weakening by a composition is the composition of weakenings:

```
_∘_ : ∀ {Ω1 Ω2 Ω3 : ICtx} → (w2 : Ω2 ⊆ Ω3) → (w1 : Ω1 ⊆ Ω2) → Ω1 ⊆ Ω3
weakenTerm∘ : ∀ {Ω1 Ω2 Ω3 τ} → (w2 : Ω2 ⊆ Ω3) (w1 : Ω1 ⊆ Ω2)
  → (A : Term Ω1 τ)
  → Id (weakenTerm A (λ {_} → w2 ∘ w1)) (weakenTerm (weakenTerm A w1) w2)
```

Another artifact of intensional equality is that we must prove that weakening by point-wise equal functions is the identity:

```
WEq : ∀ {Ω Ω'} → (f1 f2 : Ω ⊆ Ω') → Set
WEq f1 f2 = {τ : BaseType} → (x : τ ∈ _) → Id (f1 x) (f2 x)
weakenTerm-ext : ∀ {Ω Ω'} {w1 w2 : Ω ⊆ Ω'} → WEq w1 w2 →
  {τ : Type} (t : Term Ω τ) → Id (weakenTerm t w1) (weakenTerm t w2)
```

These are all the generic operations on terms that we required in our development, though we could go on to prove analogous properties of substitution, etc.

For atomic propositions, we similarly define a datatype Aprop Ω. This datatype is parametrized over a set of atomic proposition symbols Atom : Type → Set, and an atomic proposition p · t consists of an Atom A paired with a term of type A. We could have used Terms of a particular base type prop to represent atomic propositions, but for a historical accident: we had abstracted out a signature of atomic propositions before doing so for terms.

Propositions

BL_0 propositions include conjunction, disjunction, implication, universal and existential quantification, and the says modality:

$$A, B, C \quad ::= \quad P \mid A \wedge B \mid A \vee B \mid A \supset B \mid \top \\ \mid \bot \mid \forall x{:}\tau.s \mid \exists x{:}\tau.A \mid k \text{ says } A$$

In Figure 3.6, we represent this syntax in Agda. Propositions (Propo) are indexed by a context of free variables, and additionally by a *polarity* (+ or −), which will be helpful in defining a focused sequent calculus below. Because the syntax of propositions is polarized, there are two injections a− and a+ from atomic propositions Aprop to negative and positive propositions, respectively. Additionally, the shifts ↓ and ↑ include negative into positive and vice verse. The

```
data Propo : Polarity → ICtx → Set where
  _⊃_  : ∀ {Ω} → Propo+ Ω → Propo- Ω → Propo- Ω
  ∀i_  : ∀ {Ω τ} → Propo- (τ :: Ω) → Propo- Ω
  a-   : ∀ {Ω} → Aprop Ω → Propo- Ω
  ↑    : ∀ {Ω} → Propo+ Ω → Propo- Ω
  _∨_  : ∀ {Ω} → Propo+ Ω → Propo+ Ω → Propo+ Ω
  _∧_  : ∀ {Ω} → Propo+ Ω → Propo+ Ω → Propo+ Ω
  ⊥    : ∀ {Ω} → Propo+ Ω
  ⊤    : ∀ {Ω} → Propo+ Ω
  ∃i_  : ∀ {Ω τ} → Propo+ (τ :: Ω) → Propo+ Ω
  _says_ : ∀ {Ω} → Term Ω principal →
    Propo- Ω → Propo+ Ω
  a+   : ∀ {Ω} → Aprop Ω → Propo+ Ω
  ↓    : ∀ {Ω} → Propo- Ω → Propo+ Ω
```

Figure 3.6: Agda Representation of BL_0 Propositions

remaining datatype constructors correspond to the various ways of forming propositions in the above grammar. For example, the _∧_ constructor takes two terms of type Propo+ Ω and returns a term of type Propo+ Ω. The constructor ∃i (existential quantification over individuals), takes a positive proposition, in a context with one new free variable of type τ, and returns a positive proposition in the original context Ω.

We have suppressed the shifts up to this point in the paper for readability. We could suppress shifts in our Agda code by implementing a simple translation that, given an unpolarized proposition and an intended polarization of each atom, computes a polarized proposition with minimal shifts.

Proofs

Sequent calculus. Sequents in BL_0 have the form $\Omega; \Delta; \Gamma \xrightarrow{k} A$. The context Ω gives types to individual variables (e.g. it is extended by $\forall$), and the context Γ contains propositions that are assumed to be true (e.g. it is extended by $\supset$)—these are the standard contexts of first-order logic. The context Δ contains *claims*, assumptions of the form $k'\ \mathsf{claims}\ A$; claims is the judgement underlying the says connective (Garg, 2009b; Pfenning and Davies, 2001). Finally, k, the *view* of the sequent, is the principal on behalf of whom the inference is made.

The rules for says are as follows:

$$\frac{\Omega; \Delta; [] \xrightarrow{k} A}{\Omega; \Delta; \Gamma \xrightarrow{k0} k\ \mathsf{says}\ A}\ \textsc{saysR}$$

$$\frac{\Omega; \Delta, (k\ \mathsf{claims}\ A); \Gamma, (k\ \mathsf{says}\ A) \xrightarrow{k0} C}{\Omega; \Delta; \Gamma, (k\ \mathsf{says}\ A) \xrightarrow{k0} C}\ \textsc{saysL}$$

$$\frac{\Omega; (\Delta, k \ \mathsf{claims} \ A); (\Gamma, A) \xrightarrow{k0} C \qquad k \geq k0}{\Omega; (\Delta, k \ \mathsf{claims} \ A); \Gamma \xrightarrow{k0} C} \ \textsc{claimsL}$$

In order to show $k \ \mathsf{says} \ A$, one empties the context Γ of true assumptions, and reasons on behalf of k with the goal A (rule saysR). It is necessary to empty Γ because the facts in it may depend on claims by the principal k_0, which are not valid when reasoning as k. The rule saysL says that if one is reasoning from an assumption $k \ \mathsf{says} \ A$, one may proceed using a new assumption that $k \ \mathsf{claims} \ A$. Claims are used by the rule claimsL, which allows passage from a claim $k \ \mathsf{claims} \ A$ to an assumption that A is actually true. This rule makes use of a preorder on principals, and asserts that any statements made by a greater principal are accepted as true by lesser principals.

Focused sequent calculus. To help with defining a proof search procedure, we present BL_0 as a weakly-focused sequent calculus. Garg (2009b) describes both an unfocused sequent calculus and a focused proof system for FHH, a fragment of BL_0; here we give a focused sequent calculus for all of BL_0. Focusing (Andreoli, 1992b) is a proof-theoretic technique for reducing inessential non-determinism in proof search, by exploiting the fact that one can chain together certain proof steps into larger steps. In the Agda code above, we polarized the syntax of propositions, dividing them into positive and negative classes. Positive propositions, such as disjunction, require choices on the right, but are invertible on the left: a goal C is provable under assumption A^+ if and only if it is provable under the left rule's premises. Dually, negative propositions involve choices on the left but are invertible on the right. Weak focusing (Pfenning and Simmons, 2009) forces focus (choice) steps of like-polarity connectives to be chained together, but does not force inversion (pattern-matching) steps to be chained together. We use weak, rather than full, focusing because it is slightly easier to represent in Agda, and because it can sometimes lead to shorter proofs if one internalizes the identity principles (which say that A entails A)—though we do not exploit this fact in our current prover.

The polarity of $k \ \mathsf{says} \ A$ is as follows: A is negative, but $k \ \mathsf{says} \ A$ itself is positive. As a simple check on this, observe that $k \ \mathsf{says} \ A$ is invertible on the left—one can always immediately make the claims assumption—but not on the right—because saysR clears the true assumptions. For example, a policy is often of the form $k_1 \ \mathsf{says} \ A_1, \ldots k_n \ \mathsf{says} \ A_n$, with a goal of the form $k' \ \mathsf{says} \ B$. It is necessary to use claimsL to turn all propositions of the form $k \ \mathsf{says} \ A$ in Γ into claims in Δ before using saysR on the goal—if one uses saysR first, the policy would be discarded. This polarization is analogous to $\Box$ in Pfenning and Davies (2001) and to ! in linear logic (Andreoli, 1992b), which is reasonable given that says is a necessitation modality.

Our sequent calculus, presented in Figure 3.7, has three main judgements:

- Right focus: $\Omega; \Delta; \Gamma \xrightarrow{k} [A^+]$
- Left focus: $\Omega; \Delta; \Gamma \xrightarrow{k} [A^-] > C$
- Neutral sequent: $\Omega; \Delta; \Gamma \xrightarrow{k} C^-$

Here Δ consists of claims $k \ \mathsf{claims} \ A^-$ and Γ consists of positive propositions. For convenience in the Agda implementation, we break out a one-step left-inversion judgement $\Omega; \Delta; \Gamma \xrightarrow{k} A^+ >_I C$, which applies a left rule to the distinguished proposition A^+ and then reverts to a neutral sequent. The rules are a fairly simple integration of the idea of weak focusing (Pfenning and Simmons,

2009) with the focusing interpretation of says described above.

Agda Representation In Figure 3.8, we show an excerpt of the Agda representation of this sequent calculus. First, we define a record type for a Ctx, which tuples together the Ω, Δ, Γ, and k parts of a sequent—we write Θ for such a tuple. Γ is represented as a list of propositions; Δ is represented as a list of pairs of a principal and a proposition, written p claims A; k is a term of type principal. Record fields are selected by writing R.x, where the type of the record is R and the desired field is x (e.g., Ctx.rk selects the principal from a Ctx record). Note that Ctx is a dependent record: the true context, the claims context, and the view can mention the variables bound in the individual context rΩ. We write TCtx+ Ω for List (Propo+ Ω). We define several helper functions on Ctxs: sayCtx clears the Ctx of true propositions, and changes the view of the context to its second argument. ictx (not shown) is shorthand for Ctx.rΩ. addTrue and addClaim cons a true proposition onto Γ or a claim onto Δ, respectively. addVar adds a variable to Ω, and *weakens* the rest of the context.

When writing down the calculus on paper, it is obvious that extending Ω does not affect Γ or Δ; any variables bound in Ω will be bound in Ω' $\supseteq \Omega$. However, in Agda, it is necessary to explicitly coerce F Ω to F Ω' for type families F dependent on Ω. Above, we discussed weakenTerm; weakening for propositions, claims, true contexts (weakenT+), claims contexts (weakenC), and so on, are analogous.

There are 4 judgments in our weakly-focused sequent calculus; analogously, there are 4 mutually recursive datatype declarations representing these judgements in Agda, with one datatype constructor for each inference rule. We show the constructors $\forall$L (for the left focus judgement), $\exists$L and saysL (for the left inversion judgement), saysR (for the right focus judgement), and claimsL (for the neutral sequent judgement). For the most part, the rules are a straightforward transcription of the sequent calculus rules. In $\forall$L, the function substlast substitutes a term for the last variable in a proposition; we have implemented substitution for individual variables for each of the syntactic categories. In $\exists$L, it is necessary to weaken the goal with the new variable, which is tacit in on-paper presentations.

Properties Because the sequent calculus is cut-free, consistency of closed proofs is immediate:

Consistency: For all principals k, there is no derivation of
$[]; []; [] \xrightarrow{k}\uparrow \bot$.

Proof: no rule concludes $\bot$ in right focus, and in the empty context no left focus or left inversion rules apply.

Identity and cut can be proved using the usual syntactic methods, adapting Garg's proof (Garg, 2009b) for an unfocused sequent calculus to weak focusing, following Pfenning and Simmons (2009).

3.2.2 Proof Search

We have implemented a simple proof-producing theorem prover for BL_0:

prove : Nat $\rightarrow$ (θ : Ctx) $\rightarrow$ (A : Propo- (ictx θ)) $\rightarrow$ Maybe ($\theta \vdash$ A)

$$\frac{}{\Omega;\Delta;\Gamma;P^+ \xrightarrow{k} [P^+]}\ \text{INIT+} \qquad \frac{}{\Omega;\Delta;\Gamma \xrightarrow{k} [P^-] > P^-}\ \text{INIT-}$$

$$\frac{}{\Omega;\Delta;\Gamma \xrightarrow{k} [\top]}\ \top \qquad \frac{\Omega;\Delta;\Gamma \xrightarrow{k} C}{\Omega;\Delta;\Gamma \xrightarrow{k} \top >_I C}\ \top L \qquad \frac{}{\Omega;\Delta;\Gamma \xrightarrow{k} \bot >_I C}\ \bot$$

$$\frac{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+] \qquad \Omega;\Delta;\Gamma \xrightarrow{k} [B^+]}{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+ \wedge B^+]}\ \wedge\text{R} \qquad \frac{\Omega;\Delta;\Gamma, A^+, B^+ \xrightarrow{k} C}{\Omega;\Delta;\Gamma \xrightarrow{k} (A^+ \wedge B^+) >_I C}\ \wedge\text{L}$$

$$\frac{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+]}{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+ \vee B^+]}\ \vee\text{R1} \qquad \frac{\Omega;\Delta;\Gamma \xrightarrow{k} [B^+]}{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+ \vee B^+]}\ \vee\text{R2} \qquad \frac{\Omega;\Delta;\Gamma, A^+, \xrightarrow{k} C \qquad \Omega;\Delta;\Gamma, B^+, \xrightarrow{k} C}{\Omega;\Delta;\Gamma \xrightarrow{k} (A^+ \vee B^+) >_I C}\ \vee\text{L}$$

$$\frac{\Omega \vdash t : \tau \qquad \Omega;\Delta;\Gamma \xrightarrow{k} [[t/x]A^+]}{\Omega;\Delta;\Gamma \xrightarrow{k} [\exists x : \tau.A^+]}\ \exists\text{R} \qquad \frac{\Omega, x : \tau;\Delta;\Gamma, A^+ \xrightarrow{k} C}{\Omega;\Delta;\Gamma \xrightarrow{k} (\exists x : \tau.A^+) >_I C}\ \exists\text{L}$$

$$\frac{\Omega;\Delta;\Gamma \xrightarrow{k} A^-}{\Omega;\Delta;\Gamma \xrightarrow{k} [\downarrow A^-]}\ \text{BLURR} \qquad \frac{\Omega;\Delta;\Gamma, \downarrow A^- \xrightarrow{k} [A^-] > C}{\Omega;\Delta;\Gamma, \downarrow A^- \xrightarrow{k} C}\ \text{LFOC}$$

$$\frac{\Omega;\Delta;[] \xrightarrow{k} A^-}{\Omega;\Delta;\Gamma \xrightarrow{k0} [k\ \mathsf{says}\ A^-]}\ \text{SAYSR} \qquad \frac{\Omega;\Delta,(k\ \mathsf{claims}\ A^-);\Gamma \xrightarrow{k0} C}{\Omega;\Delta;\Gamma \xrightarrow{k0} (k\ \mathsf{says}\ A^-) >_I C}\ \text{SAYSL}$$

$$\frac{\Omega;\Delta;\Gamma, A^+ \xrightarrow{k} B^-}{\Omega;\Delta;\Gamma \xrightarrow{k} A^+ \supset B^-}\ \supset\text{R} \qquad \frac{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+] \qquad \Omega;\Delta;\Gamma \xrightarrow{k} [B^-] > C}{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+ \supset B^-] > C}\ \supset\text{L}$$

$$\frac{\Omega;\Delta;\Gamma \xrightarrow{k} [t/x]A^-}{\Omega;\Delta;\Gamma \xrightarrow{k} \forall x : \tau.A^-}\ \forall\text{R} \qquad \frac{\Omega;\Delta;\Gamma \xrightarrow{k} [[t/x]A^-] > C}{\Omega;\Delta;\Gamma \xrightarrow{k} [\forall x : \tau.A^-] > C}\ \forall\text{L}$$

$$\frac{\Omega;\Delta;\Gamma \xrightarrow{k} [A^+]}{\Omega;\Delta;\Gamma \xrightarrow{k} \uparrow A^+}\ \text{RFOC} \qquad \frac{\Omega;\Delta;\Gamma, A^+ \xrightarrow{k} C}{\Omega;\Delta;\Gamma \xrightarrow{k} [\uparrow A^+] > C}\ \text{BLURL}$$

$$\frac{\Omega;\Delta;(\Gamma, A^+) \xrightarrow{k} [A^+] >_I C^-}{\Omega;\Delta;(\Gamma, A^+) \xrightarrow{k} C}\ \text{LINV} \qquad \frac{\Omega;(\Delta, k\ \mathsf{claims}\ A^-);\Gamma \xrightarrow{k0} [A^-] > C \qquad k \geq k0}{\Omega;(\Delta, k\ \mathsf{claims}\ A^-);\Gamma \xrightarrow{k0} C}\ \text{CLAIMSL}$$

Figure 3.7: Weakly focused sequent calculus for BL_0

```
record Ctx : Set where
  field rΩ : ICtx
        rΓ+ : List (Propo+ rΩ)
        rΔ : List (Term rΩ principal × Propo- rΩ)   -- pairs written (k claims A)
        rk : Term rΩ principal

addTrue : (θ : Ctx) → (A : Propo Pos (ictx θ)) → Ctx
addTrue θ A =
  record {rΩ = Ctx.rΩ θ; rΓ+ = A :: (Ctx.rΓ+ θ); rΔ = Ctx.rΔ θ; rk = Ctx.rk θ}

addClaim : (θ : Ctx) → (t : Claim (ictx θ)) → Ctx
addClaim θ c =
  record {rΩ = Ctx.rΩ θ; rΓ+ = Ctx.rΓ+ θ; rΔ = c :: Ctx.rΔ θ; rk = Ctx.rk θ}

addVar : (θ : Ctx) → (A : Type) → Ctx
addVar θ τ = record {rΩ = (τ :: Ctx.rΩ θ); rΓ+ = (weakenT+ (Ctx.rΓ+ θ) iS);
                     rΔ = (weakenC (Ctx.rΔ θ) iS); rk = (weakenTerm (Ctx.rk θ) iS)}

sayCtx : (θ : Ctx) → (k : Term (Ctx.rΩ θ) principal) → Ctx
sayCtx θ k = record {rΩ = Ctx.rΩ θ; rΓ+ = []; rΔ = Ctx.rΔ θ; rk = k}


mutual
  data _⊢L_>_ : (θ : Ctx) → Propo- (ictx θ) → Propo- (ictx θ) → Set where
    ∀L : ∀ {θ τ A C} → (t : Term (ictx θ) τ) →
           θ ⊢L (substlast A t) > C →
           θ ⊢L ∀i_ {ictx θ} {τ} A > C
    ...
  data _⊢I_>_ : (θ : Ctx) → (Propo+ (ictx θ)) → Propo- (ictx θ) → Set where
    ∃L : ∀ {θ τ A C}
      → (addTrue (addVar θ τ) A) ⊢ (weakenP C iS)
      → θ ⊢I (∃e τ A) > C
    saysL : ∀ {θ k s B}
      → addClaim θ (k claims s) ⊢ C
      → θ ⊢I (k says s) > C
    ...
  data _⊢R_ : (θ : Ctx) → Propo+ (ictx θ) → Set where
    saysR : ∀ {θ k A}
      → (sayCtx θ k) ⊢ A
      → θ ⊢R (k says A)
    ...
  data _⊢_ : (θ : Ctx) → Propo- (ictx θ) → Set where
    claimsL : ∀ {θ k A C}
      → (k claims A) ∈ Ctx.rΔ θ
      → θ ⊢L A > C → k ⩾ Ctx.rk θ
      → θ ⊢ C
    ...
```

Figure 3.8: Agda representation of proofs (exceprt)

prove takes a depth bound, a context, and a proposition, and attempts to find a proof of $\theta \vdash$ A with at most the given depth. The prover is *certified*: when the prover succeeds, it returns a proof, which is guaranteed by type checking to be well-formed. When the prover fails, it simply returns None. The prover is implemented by around 200 lines of Agda code.

Our prover is quite naïve, but it suffices to prove the examples in this paper. For the most part, the prover backchains over the focusing rules. However, whereas the above sequent calculus was only weakly focused, the prover is fully focused, in that it eagerly applies invertible rules, which avoids backtracking over different applications of them. If the goal is right-invertible, the prover applies right rules. Once the goal is not right-invertible (an atom or a shift $\uparrow A^+$), the prover fully left-inverts all of the assumptions in Γ. Inverting a context Γ breaks up the positive propositions using left rules, generating a list of non-invertible contexts $\Theta_1, ..., \Theta_k$ such that, if for every i, $\Theta_i \vdash C$, then $\Theta \vdash C$. Once the sequent has been fully inverted, the prover tries right-focusing (if the goal is a shift $\uparrow A^+$) and left-focusing on all assumptions in Γ and claims in Δ, until one of these choices succeeds. The focus phases involves further backtracking over choices (e.g., which branch of a disjunction to take). The focus rules for quantifiers ($\forall E$ and $\exists I$) require guessing an instantiation of the quantifier. Our current implementation is brute-force: it simply computes all terms of a given type in a given context and tries each of them in turn—we have only considered individual types with finitely many inhabitants.

The following snippet of an auxiliary function implementing the right focus judgement gives a feel for the prover:

```
proveRight : Nat → (θ : Ctx) → (A : Propo Pos (ictx θ)) → Maybe (θ ⊢R A)
proveRight Z _ _ = None
proveRight (S n) θ (A ∨ B) =
  (map ∨R1 (proveRight n θ A)) || (map ∨R2 (proveRight n θ B))
proveRight (S n) θ (A ∧ B) =
  proveRight n θ A >>= λ x → map (λ y → ∧R x y) (proveRight n θ B)
proveRight (S n) θ ⊤ = Some ⊤R
proveRight (S n) θ ⊥ = None
proveRight (S n) θ (↓ A) = map (λ x → blurR x) (proveNeutral n θ A)
proveRight (S n) θ (k0 says A) = proveNeutral n (sayCtx θ k0) A >>= λ y → Some (saysR y)
...
```

If the depth bound is 0, fail. If the goal is a disjunction, try proving the two disjuncts. if the goal is a conjunction, prove them both. If the goal is $\top$, succeed; if it is $\bot$, fail. If the goal is $\downarrow$ A, switch to proving A as a neutral sequent. Similarly, if the goal is k0 says A, switch contexts and prove A neutrally. We write map and >>= and || for functoriality, bind, and "mplus" of Maybe; || tries the left disjunct first.

The prover achieves tolerable compile times on the small examples we have considered so far (1 to 13 seconds). If it proves too slow for some examples, we have several options: First, we can improve our implementation—e.g. by implementing unification, which will eliminate much of the branching from quantifiers, or by doing a better job of clause selection. Second, we could connect Agda with an external theorem prover, following Kariso (2010). Garg has implemented theorem prover for BL_0 in ML (Garg, 2009b), which we could integrate soundly by writing a type checker for the certificates it produces. Third, we could optimize Agda itself, by fixing

some known inefficiencies in Agda's compile-time evaluation.

3.2.3 Computations

The monadic interfaces presented in Section 3.1 are currently treated as refinement types on Haskell's IO monad, which is exposed through the Agda foreign function interface. The implementations of proof-carrying file operations simply ignore their proof arguments. fix is compiled using general recursion in Haskell. In this operational model, programs written in Aglet adhere to the security policies, but no guarantees are made about programs that can access, e.g., the raw file system operations. We discuss alternatives in Section 3.4 below.

3.3 Related Work

Aglet implements security-typed programming in the style of Aura (Jia et al., 2008), PCML5 (Avijit et al., 2010), Fine (Swamy et al., 2010), and previous work by Avijit and Harper (2007) (henceforth AH), which integrate authorization logics into functional/imperative programming languages. Our main contribution relative to these languages is to show how to support security-typed programming within an existing dependently-typed language. There are also some technical differences between these languages and ours:

First, Aura, PCML5, and AH interpret says as a lax modality, whereas BL_0 interprets it as a necessitation modality to support exclusive delegation (see Section 3.2.1); Fine uses first-order classical logic and does not directly support the says modality. The context-clearing necessitation modality is more challenging to represent than a lax modality.

Second, unlike these four languages, our language treats propositions and proofs as inductively defined data, which has several applications: In Aura, all proof-carrying primitives log the supplied proofs for later audit; the programmer could implement logged operations on top of our existing interface by writing a function

$\mathsf{toString} : \mathsf{Proof}\ \Gamma\ \mathsf{A} \to \mathsf{String}$

by recursion over proofs. Recursion over propositions is also essential for writing our theorem prover inside of Agda.

Third, our indexed monad of computations allows us to encode computation on behalf of a principal, following AH. In Aura, all computation proceeds on behalf of a single distinguished principal self. In PCML5, a program can authenticate as different principals, but the credentials are less precise: in PCML5, the program authenticates as k, whereas in AH the program acquires only the ability to su from a given k' to k—which may be a useful restriction if the program is subsequently no longer running as k'. Fine does not track authentication as a primitive notion, though it seems likely it could be encoded using an As predicate and affine types.

Fourth, in PCML5, acquire uses theorem proving to deduce consequences of the policy, whereas in our language acquire only tests whether a state-dependent atom or a statement by a principal is literally in the policy, and a separate theorem prover deduces consequences from the policy. We separate theorem proving from acquire so that we may also use the same theorem prover at compile-time to statically discharge proof obligations. PCML5 and AH make use of a theorem prover only at run-time, whereas Fine uses theorem proving only at compile-time.

Fifth, PCML5 is a language for spatially distributed authorization, where resources and policies are located at different sites on a network. We have shown how to support ML5-style spatial distribution using our indexed monad, but we leave spatial distribution of policies to future work.

Sixth, the operational semantics of both PCML5 and AH include a proof-checking reference monitor; we have not yet considered such an implementation.

Several other languages provide support for verifying security properties by type checking. For example, Fournet et al. (2007) develop a type system for a process calculus, and Bengtson et al. (2008) for F#, both of which can be used to verify authorization policies and cryptographic protocols. This work addresses important issues of concurrency, which we do not consider here. A technical difference is that, in their work, proofs are kept behind the scenes (e.g., in F7, propositions are proved by the Z3 theorem prover). In contrast, our language makes the proof theory directly available to the programmer, so that propositions and proofs can be computed with (for logging or run-time theorem proving) and so that proofs can be constructed manually when a theorem prover fails. Another example of a language that does not give the programmer direct access to the proof theory is PCAL (Chaudhuri and Garg, 2009), an extension of BASH that constructs the proofs required by a proof-carrying file system (Garg and Pfenning, 2009); proof construction is entirely automated, but sometimes inserts run-time checks.

Our indexed monad was inspired by HTT (Nanevski et al., 2006). RIF (Borgström et al., 2009) also investigates applications of indexed monads to security-typed programming, but there are some technical differences: First, RIF is a new language where refinement types (using first-order classical logic) and a refined state monad are primitive notions, whereas we embed an authorization logic and an indexed monad in an existing dependently typed language. Second, RIF's monad is indexed by predicates on an explicit representation of the system state, whereas we index by policies Γ that describe an implicit ambient state.

Many security-typed languages address the problem of enforcing information flow policies (see Abadi et al. (1999); Chothia et al. (2003) for but a couple of examples). We follow Russo et al. (2008); Swamy et al. (2010) in representing information flow using an abstract type constructor (e.g., a monad or an applicative functor). Fable (Swamy et al., 2008) takes a different approach to verifying access-control, information flow, and integrity properties, by providing a type of labelled data that is treated abstractly outside of certain policy portions of the program. This mechanism facilitates checking security properties (by choosing the labels appropriately and implementing policy functions) and proving bi-simulation properties of the programs that adhere to these policies.

DeYoung and Pfenning (2009) describe a technique for representing access control policies and stateful operations in a linear authorization logic. Our approach to verifying context invariants, as in Section 3.1.1, is inspired by their work.

The literature describes a growing body of authorization logics (Abadi, 2006, 2008; Abadi et al., 1993; DeYoung et al., 2008; Garg, 2009a,b). We chose BL_0 (Garg, 2009b), a simple logic that supports the expression of decentralized policies and whose says connective permits exclusive delegation.

Appel and Felten (1999) pioneered the use of proof-carrying authorization, in which a system checks authorization proofs at run-time. Several systems have been built using PCA (Bauer et al., 2005; Garg and Pfenning, 2009; Wobber et al., 1994). Like many security-typed languages, we use dependently typed PCA to check authorization proofs at compile-time through type checking.

3.4 Discussion

In this chapter, we have described Aglet, a library embedding security-typed programming in a dependently-typed programming language. There are many interesting avenues for future work: First, we may consider embedding an authorization logic such as full BL (Garg, 2009a) that accounts for resources that change over time.

Second, we have currently implemented the monadic computation interface on top of unguarded Haskell IO commands, which provides security guarantees for well-typed programs. To maintain security in the presence of ill-typed attackers, we may instead implement our interface using a proof-carrying run-time system such as PCFS (Garg and Pfenning, 2009). Following PCML5 (Avijit et al., 2010), we may then be able to prove a progress theorem showing that well-typed programs always pass the reference monitor. Another intriguing possibility is to formalize the operational behavior of computations directly within Agda—e.g. using an algebraic axiomatization (Plotkin and Pretnar, 2009).

Third, we have shown a few small examples of using Agda to reason about the class of contexts that is possible given a particular monadic interface. In future work, we would like to explore ways of systematizing this reasoning (e.g., by using linear logic to describe transformations between contexts, as in DeYoung and Pfenning (2009)). We would also like to use Agda to analyze global properties of a particular monadic interface (such as proving a principal can never access a resource). Once we have circumscribed the contexts generated by a particular interface, we can prove such properties by induction on BL_0 proofs.

Fourth, we have thus far shown examples of entirely static and entirely dynamic verification; we would like to consider examples that mix the two. This will require using reflection to represent Agda judgements as data, so that our theorem prover does not get stuck on open Agda terms.

To illustrate this problem, suppose we set up a call to the theorem prover as follows:

```
goal : (reviewer : Term [] (▶ principal)) →
  Proof ((▶ Prin "Admin" says a- (MaySu · (▶ Prin "Public", reviewer))) :: [])
    (a- (MaySu · (▶ Prin "Public", reviewer)))
goal reviewer = solve (prove 10)
```

That is, we would like to prove, abstractly in any reviewer, that if Admin says Public may su as the reviewer, then Public may su as the reviewer. Unfortunately, the theorem prover does not succeed in proving this goal, because of the free Agda variable reviewer: the theorem prover gets stuck at trying to compute termEq reviewer reviewer. While it is true that for all terms termEq t t computes to Some, this fact is not true definitionally.

We can solve this problem by *reflecting* the goal, which has a free Agda variable, as a goal with a free individual variable in the logic: Computation proceeds with derivability assumptions in places where it gets stuck on admissibility assumptions. In this case, we replace reviewer with an individual variable of type principal:

```
reflectedGoal :
  record {
    rΩ = principal :: [];
    rΔ = [];
    rΓ+ = (▶ Prin "Admin" says a- (MaySu · (▶ Prin "Public", ▷ i0))) :: [];
    rk = ▶ Prin "Admin"}
  ⊢ (a- (MaySu · (▶ Prin "Public", ▷ i0)))
reflectedGoal = Sums.getSome (proveNeutral 10 _ _)
```

and the theorem prover succeeds.

Next, using the substitution principal for the logic, we can instantiate a derivability assumption with an admissibility assumption to prove the original goal:

```
substLast : ∀ {Ω τ Δ Γ k A}
→ record {rΩ = τ :: Ω; rΔ = Δ; rΓ+ = Γ; rk = k} ⊢ A
→ (t : Term Ω (▶ τ))
→ record {rΩ = Ω; rΔ = substCLast Δ t;
          rΓ+ = substT+Last Γ t; rk = substTermLast k t}
  ⊢ (substlast A t)



goal : (reviewer : Term [] (▶ principal)) →
  (Proof ((▶ Prin "Admin" says a- (MaySu · (▶ Prin "Public", reviewer))) :: [])
    (a- (MaySu · (▶ Prin "Public", reviewer))))
goal reviewer = substLast reflectedGoal reviewer
```

Framework support Before moving on, it is worth summarizing what logical framework features we used in this example. We used derivability, implemented "by hand" using dependent de Bruijn indices, to represent BL_0. However, we required not just ordinary derivability, but a modal version of entailment, to support says. We used admissibility functions to implement a theorem prover, to compute types and specifications (Result and Postcondition), and to write the code that we verified using the authorization logic. We used types dependent on derivability functions (the monad indexed by propositions) and applications of admissibility functions (e.g. the computed postcondition) to make the link between code and specifications. Agda supports all of this, but implementing derivability by hand for each logic is tedious—even here, we found it useful to implement a simple logical framework to parametrize BL_0 over terms and atomic propositions. This motivates the study of richer embedded logical frameworks in Part II. But first, we present an additional example of a domain-specific logic, which illustrates a different mode of use of dependent types.

Chapter 4

Semantic Differential Privacy

The example described in this chapter was implemented jointly with Jason Reed.

Large amounts of people's information are stored in databases (medical records, your Facebook profile, ...), and mechanisms to exploit this information while preserving privacy are very important. One notion of database privacy that has been studied is *differential privacy* (Dinur and Nissim, 2003; Dwork and Nissim, 2004; Dwork et al., 2006): a mechanism is differentially private if any conclusion made from the data is almost exactly as likely if any one record is omitted from the database.

Many differentially private algorithms rely on constructing a *distance-preserving function*, and then adding some noise to it. Distance preservation ensures that the distance between outputs of a function is bounded by a constant factor of the distance between the function's inputs. A function is 1-*sensitive* if it preserves distances exactly, or, more generally, k-sensitive if the output distance is never more than k times the input distance. k-sensitive functions can be made differentially private by adding noise proportional to k. However, for programming purposes, it is necessary to consider not only distance-preserving functions on $\mathcal{R}$, but on many other types as well: pairs, with distance given by the sum of the distances of the components; sets, with edit distance; and others. This can be formalized by considering general *metric spaces*, spaces equipped with a notion of distance, and distance-preserving functions on them.

Reed and Pierce (2010) observed that the proof obligations that arise in showing that functions are distance-preserving can be packaged as a type system based on affine logic. Their language is defined in the standard syntactic manner, using a type system and operational semantics. However, this type system is *incomplete*, in the sense that there are many distance-preserving functions that cannot be written in the language, and a somewhat arbitrary collection of such operations are taken as primitives in the languages design.

We can use a dependently typed host language to solve this problem. First, we formalize the semantic definition of distance-preservation as the fundamental notion. This offers a fully certified and fully extensible type of distance-preserving functions: if you can prove that a function satisfies the semantic condition, it is an element of the type of distance-preserving functions. The affine type system is then interpreted as tactics used to construct semantic elements more conveniently—it packages up the proof obligations in a convenient way. When a new primitive is necessary, it can be proved correct in the semantics and then added to the syntax. This language

architecture is similar to NuPRL (Constable et al., 1986), where a semantics of dependent types is defined first, and the syntax consists of a particular sound but incomplete collection of proof rules. However, in NuPRL the semantics exists only on paper, whereas here we implement the semantics itself inside of a proof assistant, so that it can be programmed with directly when necessary. This example illustrates a different style of program verification than the security-typed programming example above: here, dependent types are used behind the scenes to implement a semantics and an interesting type system, but the programs written using this type system are, for the most part, simply-typed. We restrict attention to differential privacy for total programs, as semantic embeddings of general recursion require more advanced meta-language features, such as partial functions (Crary, 1998) or coinduction.

In Section 4.1 we implement metric spaces; in Section 4.2, we define an affine type system for distance-preserving functions; in Section 4.3, we show the syntax sound relative to the semantics.

4.1 Metric spaces

A metric space equips a set with a notion of distance between any two points. We will find it convenient to represent distance relationally, rather than functionally, and to represent an inequality on distance, rather than equality. Thus, the primitive notion we define is the relation $x1 \sim x2 \leqslant r$, which means that the distance between x1 and x2 is less than or equal to r. If x and y are unrelated for any r, they are considered to be infinitely distant.

A distance relation on a set satisfies *identity* if any element is related to itself at any distance. A distance relation on a set satisfies *composition* if two distances can be composed, and the distance of the composition is less than the sum of the distances—i.e. it satisfies the triangle inequality. A distance relation satisfies *weakening* if the relation respects "less than or equal to" on reals in the appropriate way.

```
Identity : (A : Set) → (A → A → PosReal → Set) → Set
Identity A _~_≤_ = ∀ {x r} → x ~ x ≤ r

Composition : (A : Set) → (A → A → PosReal → Set) → Set
Composition A _~_≤_ = ∀ {x1 x2 x3 r1 r2}
  → x1 ~ x2 ≤ r1 → x2 ~ x3 ≤ r2
  → x1 ~ x3 ≤ (r1 + r2)

Weakening : (A : Set) → (A → A → PosReal → Set) → Set
Weakening _ _~_≤_ = ∀ {x1 x2 r r'} → x1 ~ x2 ≤ r → r ≤ r' → x1 ~ x2 ≤ r'
```

The type PosReal represents positive reals, and the relation $\leqslant$ represents less-than-or-equal-to on positive reals. At present, we have *postulated* a type of real numbers with the operations and proofs we have needed for this example, which means that Agda treats this type as a free variable. An alternative would be to implement the constructive reals (Bishop, 1967).

Using these definitions, we give an Agda type representing a metric space:

```
record MetS : Set1 where
  constructor met
  field
    Carrier : Set
    _≫_~_≤_ : (x1 : Carrier) (x2 : Carrier) (r : PosReal) → Set
    mid     : Identity Carrier _≫_~_≤_
    _≫_∘_   : Composition Carrier _≫_~_≤_
    weaken  : Weakening Carrier _≫_~_≤_
open MetS public
```

A metric space is defined as a record containing a carrier type, a distance relation, and proofs that the distance relation satisfies identity, composition, and weakening. MetS is a type of level Set1 because it contains a Set, the carrier. We have named the distance relation $\_\gg\_\sim\_\leqslant\_$, which allows it to be used infix as in $M \gg x \sim y \leqslant r$, where M : MetS. The last line *opens* the record type, which also allows the field names to be used without the prefix MetS.

For example, we can define a metric space whose carrier set is the real numbers, represented by a type Real, and whose distance is implemented as usual using subtraction (-) and absolute value (|-|). Like PosRreal, this type is a postulate.

```
reals : MetS
reals = met Real
  (λ x y z → | x - y | ⩽ z)
  (⩽eql ((IdM.trans (IdM.substeq |_| -same) abs0)) pos) tri ⩽trans
```

The line discharges the proof obligations necessary for identity, composition, and weakening.

Relative to the standard definition of a metric space, this definition omits two conditions: first, we do not assume that distance is symmetric; second, we do not assume that any two elements at distance 0 are equal. We will not need either of these properties in this chapter.

4.1.1 Distance-preserving functions

If A and B are metric spaces, a distance-preserving function from A to B is an underlying function on the carriers equipped with a proof that it preserves distances, in the sense that the images of any two points are no further apart in B than the points were in A.

```
Pres : (A B : MetS) (und : Carrier A → Carrier B) → Set
Pres A B und = ∀ {x1 x2 r} → A ≫ x1 ~ x2 ⩽ r → B ≫ und x1 ~ und x2 ⩽ r
record Func (A : MetS) (B : MetS) : Set where
  constructor func
  field
    und : Carrier A → Carrier B
    pres : Pres A B und
open Func public
```

Taking the distance between two functions to be given pointwise, we show that functions themselves form a metric space.

```
Pointwise : (A B : MetS) → (f g : Func A B) (r : PosReal) → Set
Pointwise _ B f g r = ∀ {x} → B ≫ (und f x) ∼ (und g x) ⩽ r

_-os_ : MetS → MetS → MetS
A -os B = met (Func A B)
  (λ f g r → Pointwise A B f g r)
  (mid B)
  (λ d1 d2 → B ≫ d1 ∘ d2)
  (λ {x1 x2 r r'} d1 d2 → λ {x} → weaken B d1 d2)
```

Identity, composition, and weakening are inherited from B.

4.1.2 Scaling

Given a metric space, we can scale it by a positive real. It will be convenient to define this by division, rather than multiplication; the proofs of the properties use some simple reasoning about division.

```
!s : PosReal → MetS → MetS
!s f A = met (Carrier A)
  (λ x y r → A ≫ x ∼ y ⩽ (r ÷ f))
  (mid A)
  (λ {x1 _ x3 r1 r2} d1 d2 →
     IdM.subst (λ r → A ≫ x1 ∼ x3 ⩽ r) distrib÷+ (A ≫ d1 ∘ d2))
  (λ d le → weaken A d (div-weaken le))
```

4.1.3 Monoidal products

The one-point space has the trivial metric:

```
ones : MetS
ones = met Unit (λ _ _ r → Unit) _ _ _
```

The metric on pairs says that d $((x, y), (x', y'))$ = d (x, x') + d (y, y'), but we spell this fact out relationally (other notions of $\otimes$ can be defined with different metrics (Reed and Pierce, 2010)). In linear logic terms, this corresponds to multiplicative pairs ($\otimes$), as the resources are split between the components.

```
record ⊗Met {A : MetS} {B : MetS}
  (x : Carrier A × Carrier B) (y : Carrier A × Carrier B) (r : PosReal) : Set where
  constructor ⊗met
  field
    r1 : PosReal
    r2 : PosReal
    δ1 : A ≫ (fst x) ∼ (fst y) ⩽ r1
    δ2 : B ≫ (snd x) ∼ (snd y) ⩽ r2
    ι : Id (r1 + r2) r
open ⊗Met public
```

Next, we define a metric space with this distance metric. The proofs follow from the corresponding properties for A and B, with some massaging.

```
_⊗s_ : MetS → MetS → MetS
A ⊗s B = met (Carrier A × Carrier B)
  (⊗Met {A} {B})
  (λ {x r} → ⊗met 0 ·0 r (mid A) (mid B) +unit)
  (λ {x1 x2 x3 x4 x5} → comp {x1} {x2} {x3} {x4} {x5})
  (λ {p q r r'} x le →
    ⊗met (r1 x) (r2 x + gap le) (δ1 x) (weaken B (δ2 x) (positivity Refl))
      (IdM.trans
        (IdM.trans +lassoc (IdM.substeq (λ x → x + gap le) (ι x)))
        (ungap le)))
  where
  comp : Composition _ ⊗Met
  comp (⊗met _ _ d1 d2 Refl) (⊗met _ _ d3 d4 Refl) =
    ⊗met _ _ (A ≫ d1 ∘ d3) (B ≫ d2 ∘ d4) assoc/comm4
```

At this point, we could define additive binary pairs as well, but they will be a special case of what we define next.

4.1.4 Large products and sums

It turns out that it is quite simple to define product and sum metric spaces over any Agda set. For Π, the carrier is an Agda dependent function space, and the distance is given pointwise, just as with -o.

```
Πs : (A : Set) → (A → MetS) → MetS
Πs A B = met ((x : A) → Carrier (B x))
  (λ f g r → (x : A) → (B x) ≫ f x ∼ g x ⩽ r)
  (λ x → mid (B x))
  (λ d1 d2 x → (B x) ≫ (d1 x) ∘ (d2 x))
  (λ d r x → weaken (B x) (d x) r)
```

For Σ A B, the carrier is an Agda dependent pair of an element x of A and an element of the carrier of B x. The distance metric asks the A components to be equal and the B components to be the appropriate distance apart. However, because the types of the B components refer to the A components, it is necessary to choose one instance at which to make the comparison (in this case, fst p2) and coerce the other second component using the equality proof.

```
Σs : (A : Set) → (A → MetS) → MetS
Σs A B = met carrier
  mets
  (λ {x} → (Refl, mid (B (fst x))))
  (λ {x1} {x2} {x3} {x4} {x5} → comp {x1} {x2} {x3} {x4} {x5})
  (λ {x1} {x2} {r} {r'} → wkn {x1} {x2} {r} {r'})
  where
    carrier = Σ λ (x : A) → Carrier (B x)
    mets = (λ p1 p2 r →
      Σ λ (c1 : Id (fst p1) (fst p2))
        → (B (fst p2) ≫ IdM.subst (λ x → Carrier (B x)) c1 (snd p1)
                      ∼ snd p2 ⩽ r))
    comp : Composition carrier mets
    comp {x, y1} {.x, y2} {.x, y3} {r1} {r2} (Refl, p1) (Refl, p2) =
      Refl, (B x) ≫ p1 ∘ p2
    wkn : Weakening carrier mets
    wkn {x, y1} {.x, y2} {r} {r'} (Refl, p) lt = Refl, weaken (B x) p lt
```

Σ takes a function A → Set to a Set, which explains the funny notation. Intentional equality is represented by the type Id, where IdM.subst gives the usual substitution eliminator. The proofs pattern-match in the input equations, which allows the goal equations to be filled in by Refl, which in turn simplifies the obligation for the second component.

4.2 Syntax

4.2.1 Types

```
data Ty : Set where
  _-o_ : Ty → Ty → Ty
  _⊗_ : Ty → Ty → Ty
  ‘1 : Ty
  ! : PosReal → Ty → Ty
  real : Ty
  ‘Π : (A : U) → (El A → Ty) → Ty
  ‘Σ : (A : U) → (El A → Ty) → Ty
```

The first part of the syntax of types is unsurprising, consisting of constructors for functions, monoidal products, scaling, and real numbers.

Next, we exploit the fact that we are defining this language inside of a dependently typed meta-language by enriching the language with a simple form of dependent types. In particular, we consider only dependency on sets (which can also be thought of as discrete metric spaces). This allows us to avoid the question of what a type dependent on a metric space means (though we will return to a similar issue in Part III), as the domain of dependent functions and pairs is discrete. The simplest type for these constructors would be

```
‘Π‘Σ : (A : Set) → (A → Ty) → Ty
```

That is, ‘Π takes a set, and a function that picks out a Ty for every element of A, and gives back a Ty. The function here is an admissibility: the body of the ‘Π can be defined by case-analysis or recursion on the element of A, which we will exploit below.

However, for technical reasons, it is preferable to replace Set with a *universe* U, which is itself a Set. U is populated by "codes" for sets, and equipped with an elimination that decodes each code as a set:

```
data U : Set where
  Πu Σu : (A : U) → (El A → U) → U
  _∨_ : (A B : U) → U
  ⊤ ⊥ : U
  nat : U
  bool : U
  list : U → U
El : U → Set
El (Πu A B) = (x : El A) → El (B x)
El (Σu A B) = Σ λ (x : El A) → El (B x)
El (A ∨ B) = Either (El A) (El B)
El ⊤ = Unit
El ⊥ = Void
El nat = Nat
El bool = Bool
El (list A) = List (El A)
```

Here, we define a simple universe consisting of dependent functions and products, disjoint sums, natural numbers, booleans, and lists. The decoding function El recursively computes a Set from each code. Because of the dependent types, it is necessary to define U and El simultaneously, using induction-recursion (Dybjer, 2000), because the body of a Πu code is an admissibility function into codes.

There are two reasons to use U instead of abstracting over any Set. The first is that, because of predicativity, if we used Set, then Ty itself would be a Set1. As an engineering issue, we had already done much of the development before adding the dependent type constructors; and switching to a Set1 would have required rewriting some code. A more significant reason is that we will soon define a datatype indexed by Ty. Though Agda allows it, the theory of datatypes indexed non-uniformly by Set1's has not been studied, and while no problems are known, the universe approach allows us to play it safe.

4.2.2 Contexts

Contexts are lists of assumptions A [r], meaning A is true with *sensitivity* r. The sensitivity is a scaling factor, equivalent to ! r A. A function A -o B internalizes a term of type B with free variable A [1], and is thus said to be 1-sensitive. As a type, a context A1 [r1], ..., An [rn] is equivalent to (! r1 A1) ⊗ ... ⊗ (! rn An).

```
record Item : Set where
  constructor _ [_]
  field
    ty : Ty
    sen : PosReal
open Item public

Ctx = List Item
```

We will sometimes apply a function to the sensitivities in the context, in particular scaling them with multiplication and division:

```
mapsens : (PosReal → PosReal) → Ctx → Ctx
mapsens f = ListM.map (λ Ar → ty Ar [f (sen Ar)])

_*c_ : PosReal → Ctx → Ctx
r *c Γ = mapsens (λ x → x * r) Γ

_÷c_ : Ctx → PosReal → Ctx
Γ ÷c r = mapsens (λ x → x ÷ r) Γ
```

If two contexts Γ1 and Γ2 have the same types, their *join* is a third context with the sum of their sensitivities:

```
  -- Γ1 , Γ2 = Γ3
data Join : Ctx → Ctx → Ctx → Set where
  Done : Join [] [] []
  Cons : ∀ {Γ1 Γ2 Γ3 r1 r2 A} → Join Γ1 Γ2 Γ3
    → Join (A [r1] :: Γ1) (A [r2] :: Γ2) (A [r1 + r2] :: Γ3)
```

4.2.3 Terms

First, we define a datatype of constants, indexed by their types:

```
data Const : Ty → Set where
  cmpswp : Const (real -o real -o real ⊗ real)
  rsplit : Const (real -o real ⊗ real)
```

cmpswp compares two real numbers and puts them into increasing order; it is 1-sensitive. It is necessary to take compare-and-swap as a primitive because it is not possible to give a general

1-sensitive comparison on reals, such as positive : real -o bool. Consider what it means for such a function to preserve distances: given any two reals of distance at most r, the booleans it produces are of distance at most r. But the space of booleans is discrete, so two booleans are only non-infinitely-distant if they are identical. So a 1-sensitive function must take e.g. -1 and 1, which are not infinitely far apart, to the same boolean. However, compare-and-swap is 1-sensitive, because it puts the two numbers into increasing order, but does not tell you whether it swapped them or not.

Similarly, rsplit injects its argument into the left or the right component of the pair, depending on whether it is positive or negative, and leaves the other component 0, but it does not tell you whether it was positive or negative.

Next, we define a datatype of typed terms in context—i.e. the typing derivations of the language—in Figure 4.1. Constants can be used at their stated types with the rule const. Hypotheses can be used with the rule var if they are assumed at sensitivity greater than 1. This rule builds in two kinds of weakening: first, the remaining variables in Γ may go unused, and second, any remaining sensitivity above 1 may be discarded. The rules lam and app for -o are the standard rules modified to track sensitivity in the following manner: in the introduction rule, the new assumption is assumed 1-sensitive; in the elimination rule, the sensitivities used in the function position and in the argument position are joined. Pair introduction pair is similar. Pair elimination letpair says that (1) if you can prove A $\otimes$ B from Γ and (2) assuming A and B with sensitivity r you can prove C from Δ, then you can prove C using Γ scaled by r and Δ. The reason this rule is a bit complicated is that it builds in a use of the substitution principle, which has the form

$$\text{If } \Gamma \vdash A \text{ and } \Delta, A[r] \vdash C \text{ then } (r * \Gamma), \Delta \vdash C$$

or, writing $\Gamma \vdash A[r]$ to mean $\Gamma \div r \vdash A$, it can equivalently be stated as

$$\text{If } \Gamma \vdash A[r] \text{ and } \Delta, A[r] \vdash C \text{ then } \Gamma, \Delta \vdash C.$$

Similarly, the rule bang for ! introduction could equivalently be written with a division in the premise, rather than a multiplication in the conclusion. The rule let! for ! elimination builds in a substitution, similarly to letpair; here the s is the sensitivity from the substitution principle. The rules dlam and dapp use admissibility functions to inherit the type from Agda: a ‘Π is introduced by giving an Agda function that yields a derivation of $\Gamma \vdash$ B x for every element x of A, and eliminated by choosing such an element. Dually, ‘Σ is introduced by such a choice and eliminated by hypothesizing the first component of the pair, using an admissibility function, and the second component, using a derivability assumption. dletpair also manipulates the sensitivities in the same way as the substitution principle.

4.2.4 Derived forms

As a simple example, we show that binary additives ($\oplus$ and &) are definable using ‘Σ and ‘Π. First, we define if for booleans:

```
if_then_else : {C : Set} → Bool → C → C → C
if True then x else y = x
if False then x else y = y
```

```
data _⊢_ : Ctx → Ty → Set where
  const : ∀ {Γ A}
        → Const A
        → Γ ⊢ A
  var : ∀ {Γ A r}
      → (A [ r ]) ∈ Γ → 1 ·0 ⩽ r
      → Γ ⊢ A
  lam : ∀ {Γ A B}
      → (A [ 1 ·0 ] :: Γ ⊢ B)
      → Γ ⊢ A -o B
  app : ∀ {Γ1 Γ2 Γ3 A B}
      → Γ1 ⊢ A -o B → Γ2 ⊢ A → Join Γ1 Γ2 Γ3
      → Γ3 ⊢ B
  ⟨⟩ : ∀ {Γ}
      → Γ ⊢ ‘1
  pair : ∀ {Γ1 Γ2 Γ3 A B}
      → Γ1 ⊢ A → Γ2 ⊢ B → Join Γ1 Γ2 Γ3
      → Γ3 ⊢ A ⊗ B
  letpair : ∀ {Γ A B C Δ r Γ'}
      → Γ ⊢ A ⊗ B → (A [ r ]) :: (B [ r ]) :: Δ ⊢ C → Join (r *c Γ) Δ Γ'
      → Γ' ⊢ C
  bang : ∀ {Γ A r}
       → Γ ⊢ A
       → (r *c Γ) ⊢ ! r A
  let! : ∀ {Γ1 Γ2 Γ3 A C r s}
       → Γ1 ⊢ ! r A → (A [ r * s ] :: Γ2) ⊢ C → Join (s *c Γ1) Γ2 Γ3
       → Γ3 ⊢ C
  dlam : ∀ {Γ A B}
       → ((x : El A) → Γ ⊢ (B x))
       → Γ ⊢ ‘Π A B
  dapp : ∀ {Γ A B}
       → Γ ⊢ ‘Π A B → (t : El A)
       → Γ ⊢ (B t)
  dpair : ∀ {Γ A B}
       → (t : El A) → Γ ⊢ (B t)
       → Γ ⊢ ‘Σ A B
  dletpair : ∀ {r Γ1 Γ2 Γ3 A B C}
      → Γ1 ⊢ ‘Σ A B → ((x : El A) → ((B x) [ r ] :: Γ2) ⊢ C) → Join (r *c Γ1) Γ2 Γ3
      → Γ3 ⊢ C
```

Figure 4.1: Typing rules for differential privacy

Next, sums are encoded as usual in dependent type theory: as a boolean tag bit, followed by an element of either A or B, depending on whether the tag bit is true for false:

```
_⊕_ : Ty → Ty → Ty
A ⊕ B = 'Σ bool (λ b → if b then A else B)
```

The intro rules simply tag the term with the appropriate bit:

```
inl : ∀ {Γ A B} → Γ ⊢ A → Γ ⊢ (A ⊕ B)
inl e = dpair True e
inr : ∀ {Γ A B} → Γ ⊢ B → Γ ⊢ (A ⊕ B)
inr e = dpair False e
```

while the elimination rule checks the tag and defers to the appropriate branch:

```
case : ∀ {Γ1 Γ2 Γ3 A B C r}
  → Γ1 ⊢ (A ⊕ B)
  → (A [ r ] :: Γ2) ⊢ C → (B [ r ] :: Γ2) ⊢ C
  → Join (r *c Γ1) Γ2 Γ3
  → Γ3 ⊢ C
case {Γ1} {Γ2} {Γ3} {A} {B} {C} {r} e e1 e2 = dletpair e b where
  b : (x : Bool) → (if x then A else B) [ r ] :: Γ2 ⊢ C
  b True = e1
  b False = e2
```

4.3 Soundness

4.3.1 Combinators

Instead of translating to metric spaces directly, we give a syntax that is closer to the metric spaces, and use that as an intermediary. The reason for this is technical: the translation of a natural deduction rule typically involves several of the primitive operations on the corresponding metric spaces—e.g. because the natural deduction rule plumbs the context through each rule. For example, we may have to apply an operation that reassociates the context, which is represented as tuple using $\otimes$:

```
lassoc : ∀ {A B C} → (A ⊗ (B ⊗ C)) ⇒ ((A ⊗ B) ⊗ C)
```

However, when this rule is phrased in terms of the semantics, it is difficult for Agda to figure out the implicit arguments, because $\otimes$s is not treated as injective. On the other hand, when this rule is phrased in terms of the syntax, $\otimes$ is a datatype constructor and therefore injective. Thus, we can simplify the syntax of the translation considerably by going through this intermediate step,

so that when we are composing together these combinators, the implicit arguments are filled in automatically.

The combinators are defined as follows:

```
data _⇒_ : Ty → Ty → Set where
  ⇒c : ∀ {A} → Const A → ‘1 ⇒ A
  id : ∀ {A} → A ⇒ A
  _∘_ : ∀ {A B C} → B ⇒ C → A ⇒ B → A ⇒ C
  abs : ∀ {A B C} → (A ⊗ B) ⇒ C → A ⇒ (B -o C)
  eval : ∀ {A B} → ((A -o B) ⊗ A) ⇒ B
  wk : ∀ {A B C} → B ⇒ C → (A ⊗ B) ⇒ C
  _⊗m_ : ∀ {A B C D} → A ⇒ B → C ⇒ D → A ⊗ C ⇒ B ⊗ D
  tw : ∀ {A B} → (A ⊗ B) ⇒ (B ⊗ A)
  oneδ : ∀ {A} → A ⇒ (A ⊗‘1)
  onei : ∀ {A} → A ⇒‘1
  rassoc : ∀ {A B C} → ((A ⊗ B) ⊗ C) ⇒ (A ⊗ (B ⊗ C))
  lassoc : ∀ {A B C} → (A ⊗ (B ⊗ C)) ⇒ ((A ⊗ B) ⊗ C)
  dabs : ∀ {Γ A B} → ((x : El A) → Γ ⇒ B x) → Γ ⇒‘Π A B
  dapp : ∀ {A B} → (t : El A) →‘Π A B ⇒ B t
  dpair : ∀ {A B} → (t : El A) → B t ⇒‘Σ A B
  dsplit : ∀ {A B C} → ((x : El A) → B x ⇒ C) →‘Σ A B ⇒ C
  rassocΣ : ∀ {A B C} → ((‘Σ A B) ⊗ C) ⇒ (‘Σ A (λ x → B x ⊗ C))
  lassocΣ : ∀ {A B C} → (‘Σ A (λ x → B x ⊗ C)) ⇒ ((‘Σ A B) ⊗ C)
  !i : ∀ {A r} → r ⩽ 1 ·0 → A ⇒ ! r A
  !e : ∀ {A r} → 1 ·0 ⩽ r → ! r A ⇒ A
  !split : ∀ {A r s} → (! (r + s) A) ⇒ ((! r A) ⊗ (! s A))
  !i2 : ∀ {A r} →‘1 ⇒ A →‘1 ⇒ ! r A
  !f : ∀ {A B r} → A ⇒ B → ! r A ⇒ ! r B
  !co : ∀ {A s r} → ! r (! s A) ⇒ ! (s * r) A
  !coi : ∀ {A s r} → ! (s * r) A ⇒ ! r (! s A)
  !⊗d : ∀ {A B r} → ! r (A ⊗ B) ⇒ (! r A) ⊗ (! r B)
  !⊗di : ∀ {A B r} → (! r A) ⊗ (! r B) ⇒ ! r (A ⊗ B)
  !Σd : ∀ {A B r} → ! r (‘Σ A B) ⇒‘Σ A (λ x → ! r (B x))
  !Σdi : ∀ {A B r} →‘Σ A (λ x → ! r (B x)) ⇒ ! r (‘Σ A B)
```

From these we can derive weakening (on the right) and interchange:

```
wk' : ∀ {A B C} → B ⇒ C → (B ⊗ A) ⇒ C
wk' f = wk f ∘ tw

intch : ∀ {A B C D} → ((A ⊗ B) ⊗ (C ⊗ D)) ⇒ ((A ⊗ C) ⊗ (B ⊗ D))
intch = lassoc ∘ (id ⊗m (rassoc ∘ (tw ⊗m id) ∘ lassoc)) ∘ rassoc
```

4.3.2 Source to combinators

First, we define the interpretation of a context as a type, as discussed above.

```
⟦_⟧ct : Ctx → Ty
⟦ A [ r ] :: Γ ⟧ct = ⟦ Γ ⟧ct ⊗ (! r A)
⟦ [] ⟧ct = ‘1
```

Next, we prove two lemmas about contexts. First, the meta-operation of multiplying by a sensitive has the same effect as ! (we need only one direction). Second, the join of two contexts is the same as their tensor (but again we only need one direction):

```
ctx! : ∀ {Γ r} → ⟦ r *c Γ ⟧ct ⇒ ! r ⟦ Γ ⟧ct
ctx! {[]} = !i2 id
ctx! {B [ s ] :: Γ} = !⊗di ∘ (ctx! {Γ} ⊗m !coi)
join-lemma : ∀ {Γ1 Γ2 Γ} → Join Γ1 Γ2 Γ → ⟦ Γ ⟧ct ⇒ (⟦ Γ1 ⟧ct ⊗ ⟦ Γ2 ⟧ct)
join-lemma Done = oneδ
join-lemma (Cons j) = intch ∘ (join-lemma j ⊗m !split)
```

Next, we interpret variables as a series of weakenings around a use of !e:

```
varf : ∀ {Γ A r} → (A [ r ]) ∈ Γ → 1 ·0 ⩽ r → ⟦ Γ ⟧ct ⇒ A
varf i0 le = wk (!e le)
varf {B [ r ] :: Γ} (iS y) le = wk' (varf y le)
```

Finally, the translation to combinators is an exercise in programming by type-checking. We recommend reading this in the companion code in Agda, so that you can investigate the types of each clause.

```
⟦_⟧1 : ∀ {Γ A} → Γ ⊢ A → ⟦ Γ ⟧ct ⇒ A
⟦ var i le ⟧1 = varf i le
⟦ const c ⟧1 = ⇒c c ∘ onei
⟦ lam e ⟧1 = abs (⟦ e ⟧1 ∘ (id ⊗m !i ⩽refl))
⟦ app e e' j ⟧1 = eval ∘ (⟦ e ⟧1 ⊗m ⟦ e' ⟧1) ∘ join-lemma j
⟦ ⟨⟩ ⟧1 = onei
⟦ pair e e' j ⟧1 = (⟦ e ⟧1 ⊗m ⟦ e' ⟧1) ∘ join-lemma j
⟦ letpair {Γ} e e' j ⟧1 = ((⟦ e' ⟧1 ∘ lassoc) ∘ tw ∘ (tw ⊗m id)) ∘
                          (((!⊗d ∘ !f ⟦ e ⟧1) ∘ ctx! {Γ}) ⊗m id) ∘ join-lemma j
⟦ bang {Γ} x ⟧1 = !f ⟦ x ⟧1 ∘ ctx! {Γ}
⟦ let! {Γ1} e e' j ⟧1 = ⟦ e' ⟧1 ∘ (id ⊗m (!co ∘ (!f ⟦ e ⟧1) ∘ ctx! {Γ1})) ∘ tw ∘ join-lemma j
⟦ dlam e ⟧1 = dabs (λ x → ⟦ e x ⟧1)
⟦ dapp e t ⟧1 = dapp t ∘ ⟦ e ⟧1
⟦ dpair t e ⟧1 = (dpair t) ∘ ⟦ e ⟧1
⟦ dletpair {r} {Γ1} {Γ2} {Γ3} e1 e2 j ⟧1 = dsplit (λ x → ⟦ e2 x ⟧1 ∘ tw)
  ∘ ((rassocΣ) ∘ (!Σd ⊗m id))
  ∘ (!f ⟦ e1 ⟧1 ∘ ctx! {Γ1} ⊗m id {⟦ Γ2 ⟧ct}) ∘ join-lemma j
```

4.3.3 Combinators to metric spaces

First, we interpret types using the corresponding metric space constructors:

```
⟦_⟧ : Ty → MetS
⟦ A -o B ⟧ = ⟦ A ⟧ -os ⟦ B ⟧
⟦ A ⊗ B ⟧ = ⟦ A ⟧ ⊗s ⟦ B ⟧
⟦ real ⟧ = reals
⟦ ! r A ⟧ = !s r ⟦ A ⟧
⟦'1 ⟧ = ones
⟦'Π A B ⟧ = Πs (El A) (λ x → ⟦ B x ⟧)
⟦'Σ A B ⟧ = Σs (El A) (λ x → ⟦ B x ⟧)
⟦_⟧c : Ctx → MetS
⟦ Γ ⟧c = ⟦ ⟦ Γ ⟧ct ⟧
```

We also compose this with the type interpretation of contexts.

The interpretation of combinators as metric spaces has the following type:

```
⟦_⟧₂ : ∀ {A B} → A ⇒ B → Func ⟦ A ⟧ ⟦ B ⟧
```

A combinator term from A to B is interpreted as a distance-preserving function from ⟦ A ⟧ to ⟦ B ⟧. The implementation is about 90 lines of code, which we excerpt briefly here. The underlying functions are what one would expect if you ignore the metrics and read the types of the combinators intuitionistically. The proofs of distance-preservation are not difficult, but they do involve a bit of arithmetic and equality manipulation. The case for eval is typical:

```
⟦ eval {A} {B} ⟧₂ = record {
  und = λ fa → und (fst fa) (snd fa);
  pres = (λ {fa1 fa2 r} d →
    IdM.subst (λ z → ⟦ B ⟧ ≫ und (fst fa1) (snd fa1) ∼ und (fst fa2) (snd fa2) ⩽ z)
      (+twist (ι d))
      (⟦ B ⟧ ≫ pres (fst fa1) (δ2 d) ∘ (δ1 d)))}
```

The underlying function is given by function application. The proof of distance preservation uses (1) the distance between the two functions on any single argument ($\delta 1$ d), (2) the distance between the arguments ($\delta 2$ d), (3) distance preservation for (fst fa1), and (4) an equality coercion using a proof that addition is commutative (+twist).

Primitives, like cmpswp, are interpreted by implementing the desired function on the carrier—in this part of the semantics, one is free to use boolean-valued less-than, etc.—and then doing the arithmetic to prove the implementation preserves distances.

Finally, we tie it all together by composition:

```
⟦_⟧₁₂ : ∀ {Γ A} → Γ ⊢ A → Func ⟦ Γ ⟧c ⟦ A ⟧
⟦ e ⟧₁₂ = ⟦ ⟦ e ⟧₁ ⟧₂
```

4.4 Examples

First, we need a type of lists:

```
rlist : Ty
nil : ∀ {Γ} → Γ ⊢ rlist
cons : ∀ {Γ} → Γ ⊢ (real -o rlist -o rlist)
listrec : ∀ {r Γ1 Γ2 Γ3 C} → Γ1 ⊢ rlist
  → Γ2 ⊢ C
  → (real [r] :: C [r] :: Γ2 ⊢ C)
  → Join (r *c Γ1) Γ2 Γ3
  → Γ3 ⊢ C
```

This type can either be added to the language and its semantics, or implemented using Σ-types and recursion as `Σ nat (λ n → vec n real) where vec 0 A = '1 and vec (S n) A = A ⊗ vec A n.

As an example, we implement a function to insert a real number in order into a list of real numbers, which is assumed to be sorted in increasing order. The type system proves that this function is 1-sensitive; Reed and Pierce (2010) use this function to define insertion sort, which shows that sorting can be used in a 1-sensitive function. In concrete syntax, insert is implemented as follows:

```
insert : rlist -o real -o rlist
insert = λ l →
  listrec l of
    [] → λ new → (cons new [])
    x :: r → λ new →
      let (smaller, bigger) = cmpswap x new in smaller :: r bigger
```

insert is 1-sensitive because it uses variables linearly, and relies on only one primitive, cmpswap, which is proved sound in the semantics.

To use Agda as a type checker for this term, we must first translate the concrete syntax into abstract syntax, and the variables into de Bruijn form:

```
insert : [] ⊢ rlist -o real -o rlist
insert =
  lam (listrec (var i0 ⩽refl)
    (lam (app (app cons (var i0 ⩽refl) ?) nil ?))
    (lam (letpair (app (app (const cmpswp) (var i0 ⩽refl) ?) (var (iS i0) ⩽refl) ?)
      ((app (app cons (var (iS i0) ⩽refl) ?)
        (app (var (iS (iS (iS (iS i0)))) ⩽refl) (var i0 ⩽refl) ?) ?)) ?)) ?)
```

Next, we must annotate each rule with more than one premise with a proof of Join, which shows how to split the context:

```
insert : [] ⊢ rlist -o real -o rlist
insert = lam (listrec (var i0 ⩽refl)
  (lam (app (app cons
    (var i0 ⩽refl)
    (Cons' right (Cons' nw Done)))
    nil (Cons' left (Cons' nw Done))))
  (lam (letpair (app (app (const cmpswp)
    (var i0 ⩽refl)
    (Cons' right (Cons' nw (Cons' nw (Cons' nw Done)))))
      (var (iS i0) ⩽refl)
      (Cons' left (Cons' right (Cons' nw (Cons' nw Done)))))
    ((app (app cons
        (var (iS i0) ⩽refl)
        (Cons' nw (Cons' right (Cons' nw (Cons' nw (Cons' nw (Cons' nw Done)))))))
      (app (var (iS (iS (iS (iS i0)))) ⩽refl)
        (var i0 ⩽refl)
        (Cons' right (Cons' nw (Cons' nw (Cons' nw (Cons' left (Cons' nw Done)))))))
      (Cons' right (Cons' left (Cons' nw (Cons' nw (Cons' right (Cons' nw Done))))))))
    (Cons' left1 (Cons' left1 (Cons' right1 (Cons' nw1 Done))))))
  (Cons' left1 Done))
```

In this code, we never split a sensitivity: when splitting a variable A [r], we assume A [r] in one branch and A [0] in the other. Thus, the proofs of Join rely on only a simple collection of arithmetic proofs, corresponding to whether the variable goes into the left branch, the right branch, or nowhere (for variables whose sensitivity is already 0). To give these directions, we use left : Id (1 + 0) 1 and right : Id (0 + 1) 1 and nw : Id (0 + 0) 0. left1, right1, and nw1 additionally cancel a multiplication by 1, which arises in elimination rules that choose an arbitrary sensitivity for the cut formula. We also use a derived version of Cons, which abstracts the constraint that the result sensitivity is the sum of the two inputs into an equality premise:

```
Cons' : ∀ {Γ1 Γ2 Γ3 r1 r2 r3 A} → Id (r1 + r2) r3 → Join Γ1 Γ2 Γ3
  → Join (A [r1] :: Γ1) (A [r2] :: Γ2) (A [r3] :: Γ3)
Cons' Refl p = Cons p
```

4.5 Discussion

In this chapter, we have an implemented an semantic approach to a programming language for differential privacy: we implemented metric spaces and distance preserving functions and mechanized the translation from Reed and Pierce (2010)'s linear-logic-based type system into this semantics. The resulting language is easily extensible by new primitives, if they can be proved sound semantically. The operational behavior of programs is inherited from the host language: the semantics also functions as a compiler from the affine system to Agda. We are currently in the process of extending this code with more type constructors, such as Reed and Pierce (2010)'s

monad of probabilistic computations, and primitives. We hope to be able to verify the solution to Netflix's privacy bug, where they released anonymized movie rental records that could nonetheless be correlated with other databases, such as IMDB (McSherry and Mironov, 2009; Narayanan and Shmatikov, 2008).

What tools have we used in this implementation? First, we exploited admissibility functions from the meta-language in a crucial way: a distance-preserving function is a function equipped with a proof that it preserves distances. This requires dependency on admissibility functions as well: this example would not be possible in Twelf, Delphin, or Beluga, which either do not have a type of admissibility functions, or do not allow dependency on them. Second, we exploited the fact that we were working in a dependently typed host language to add dependent types (over sets) to the language. Thus far, we have used these dependent types as generalized additives, though we could also explore using them for expressing, e.g., data structure invariants in differentially private code. Because of Π and Σ, the definition of the syntax/typing derivations of the language uses a mix of admissibility and derivability: e.g. the premise of dletpair binds two assumptions, one admissibility and the other derivability. We explore such mixed definitions in Part II.

It is possible, if tedious, to verify programs by writing them directly as an element of the source language datatype. Agda's implicit argument mechanism can already be used to infer types and contexts, but the representation is cluttered by (a) the de Bruijn representation of variables and (b) the context splitting annotations necessary for the affine logic. We speculate on a solution to (a) below; for the latter, we could implement *resource inference*, to infer the Join annotations, analogously to the theorem prover in the previous chapter. Given these improvements, writing elements in the Agda datatype of well-typed terms would be quite close to the desired concrete syntax.

Categorical Interpretation Another reason we have presented this example is that it foreshadows the categorical interpretation of type theory that we present in Part III. The type MetS of metric spaces is in fact the specification of an enriched category, with the Carrier being the set of objects, and the distance relation the set of morphisms (enriched by distance). Distance-preserving functions are analogous to functors. The description of the various type constructors has a similar feel to construction of a Cartesian closed structure in Cat. The fact that we have not required distance to be symmetric foreshadows the directed type theory described below.

Part II

Mixing Derivability and Admissibility

Chapter 5

An Embedded Logical Framework

The research described in this chapter was conducted jointly with Robert Harper, and published in ICFP 2009 (Licata and Harper, 2009).

The above examples motivate providing a generic implementation of derivability inside of MLTT. One option would be to implement a well-known logical framework, such as LF. However, this would not account for inductive definitions that mix derivability and admissibility assumptions. Admissibility premises are useful because they permit infinitely branching derivations, as in the differential privacy example above, and allow certain forms of *side conditions*, such as negated premises (as $J \vDash \mathsf{false}$). Thus, we instead take the opportunity to explore a new logical framework that allows such interaction. In this chapter, we focus on the hypothetical judgement, precluding dependency on assumptions. Because of this restriction to "non-dependent judgements," our examples focus on simply-typed programming with abstract syntax. Our goal is to demonstrate that

> *It is possible to implement, within a dependently typed programming language, a logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.*

As we discussed briefly in Chapter 2, the reason that mixing admissibility and derivability is difficult is that rules with admissibility premises are not necessarily pure, and, consequently, the structural properties do not necessarily hold. For example, in logical frameworks such as LF, it is always possible to weaken a value of type J to $L \vdash J$. However, this is not necessarily possible when J itself is an admissibility. The reason is that we interpret an admissibility in a derivability context, $\Psi \vdash (J \vDash K)$, as essentially the same thing as an admissibility between derivabilities: $(\Psi \vdash J) \vDash (\Psi \vdash K)$. Now, suppose we are given a function f of type $\Psi \vdash (J \vDash K)$, and we try to weaken this to a function of type $(\Psi, L) \vdash (J \vDash K)$. This requires an admissibility function from $(\Psi, L) \vdash J$ to $(\Psi, L) \vdash K$. Since f is a black box, we can only hope to achieve this by pre- and post-composing with appropriate functions. The post-composition must take $\Psi \vdash K$ to $\Psi, L \vdash K$, which is a recursive application of weakening. However, the pre-composition has a contravariant flip: we require *strengthening* $(\Psi, L) \vdash J$ to $\Psi \vdash J$ in order to call f. Such a strengthening function does not in general exist, because the derivation of J might be that assumption of L. Similarly, substitution of terms for variables is not necessarily possible, because substitution requires weakening.

As a concrete example, consider a closed admissibility function of type $\cdot \vdash (\mathsf{exp} \vDash \mathsf{exp})$, which is defined by case-analysis over closed λ-calculus expressions, giving cases for functions and applications—but *not* for variables, because there are no variables in the empty context. Weakening such a function to type $\mathsf{exp} \vdash (\mathsf{exp} \vDash \mathsf{exp})$ *enlarges* its domain, asking it to handle cases that it does not cover. Another example, which is particularly stark, is weakening $L \vDash \mathsf{false}$, which refutes the existence of any terms of type L, to $L \vdash (L \vDash \mathsf{false})$, which has one such term to account for.

Our solution to this problem rests on the observation that there is nothing about the inductive definition of derivability that requires the structural properties to hold. We saw this in the simple logical frameworks described in Section 2: the identity principle is taken as a rule, but the remaining structural properties are proved admissible. Thus, we may give a definition of a framework that allows both admissibility and *pre-derivability*. Pre-derivability captures the notion of a scoped assumption, which can be used via the identity principal, but does not necessarily satisfy the remaining structural properties. Then, we analyze circumstances under which the structural properties hold, in which case pre-derivability is actually derivability.

More concretely, we implement, in Agda, a logical framework with two function-like type constructors, $\supset$ (for admissibility) and $\Rightarrow$ (for pre-derivability). $A \supset B$ classifies Agda functions, while $D \Rightarrow A$ classifies *values of type A with a free scoped assumption of type D*. In some cases, $D \Rightarrow A$ represents a derivability, and therefore determines a function given by substitution, but in some cases it does not. However, rather than leaving it to the programmer to implement the structural properties, we observe that they *are* in fact definable generically, not for every type $D \Rightarrow A$, but under certain conditions on the types D and A. For example, returning to our failed attempt to weaken $A \supset B$ above, if variables of type D could never appear in terms of type A, then the required strengthening operation would exist. As a rough rule of thumb, one can weaken with types that do not appear to the left of a computational arrow in the type being weakened, and similarly for substitution. Our framework implements the structural properties generically but conditionally, providing programmers with the structural properties "for free" in many cases. We implement $\Rightarrow$ with well-scoped de Bruijn indices, but another first-order representation of derivability with explicit contexts could be used instead.

Our framework is implemented as a universe in Agda (we saw a simple example of a universe in Chapter 4). This means that we (a) give a syntax for the types of the framework and (b) give a function mapping the types of our language to certain Agda types; the programs of the framework are then the Agda programs of those types. This implementation strategy allows us to reuse the considerable implementation effort that has gone into Agda, and to exploit generic programming within dependently typed programming (Altenkirch and McBride, 2003) to implement the structural properties. Additionally, it permits programs written using our framework to interact with existing Agda code.

In particular, we define a universe of *contextual* types, which permit inference rules to be written in the *local* form discussed in Chapter 2: the context of scoped assumptions need not be mentioned explicitly in rules, but is passed in from the outside. This often permits more concise specifications. Semantically, a type in the universe is interpreted as a function from contexts to Agda types. For example, the contextual type $A \otimes B$, is interpreted as the function $\lambda\Psi.\, \langle\Psi\rangle\, A \times \langle\Psi\rangle\, B$, where $\langle\Psi\rangle\, A$ stands for the interpretation of A at Ψ, and $\times$ is pairing in Agda. The universe types are thus "point-free" combinators that generate functions from con-

texts to types, without mentioning the context argument explicitly. For example, $D \Rightarrow A$ is interpreted as $\lambda\Psi.\, \langle \Psi, D \rangle\, A$—it extends the current context without mentioning it by name. That said, our framework supplies a rich enough set of combinators that contexts can be mentioned explicitly when this is helpful—for example, the contextual type $[\Psi]A$ represents a constant function $\lambda\_.\, \langle \Psi \rangle\, A$ which interprets A relative to the given Ψ, ignoring the context supplied by the interpretation.

Our framework implements a variety of structural properties for the universe, including weakening, substitution, exchange, contraction, and subordination-based strengthening (Virga, 1999), all using a single generic map function for datatypes that mix binding and computation. The structural properties' preconditions are defined computationally, so that our framework can discharge these conditions automatically in many cases. This gives the programmer free access to weakening, substitution, etc. (when they hold).

In Section 5.1, we introduce our language and its semantics. In Section 5.2, we present examples. We program a variety of examples, and demonstrate that we can express detailed invariants about variable usage in a program's type while still writing clean and clear code. For example, we implement normalization-by-evaluation (Berger and Schwichtenberg, 1991; Martin-Löf, 1975) for the untyped λ-calculus, an example considered in FreshML by Shinwell et al. (2003). Our version of this algorithm makes essential use of a datatype mixing binding and computation, and our type system verifies that evaluation maps closed terms to closed terms. In Section 5.3, we discuss the structural properties. In Sections 5.4, we discuss related work.

5.1 Language Definition

5.1.1 Types

The grammar for the types of our language is as follows:

Defined atoms	D	$::=$	$\ldots$
Var. Types	C	$::=$	(a subset of D)
Contexts	Ψ	$::=$	$[] \mid (\Psi, C)$
Types	A	$::=$	$`0 \mid `1 \mid A \otimes B \mid A \oplus B \mid \mathsf{list} A \mid A \supset B$
			$\mathsf{D}^{+}\, D \mid C\# \mid \Psi \Rightarrow^{*} A \mid \Box A$
			$\forall_{\mathsf{c}}\, \psi.A \mid \exists_{\mathsf{c}}\, \psi.A \mid \forall_{\leqslant C} A \mid \exists_{\leqslant C} A$

The language is parametrized by a class of defined atoms D, which are the names of datatypes. A subset of these names are variable types, which are allowed to appear in contexts. This distinguishes certain types C which may be populated by variables from other types D which may not. This definition of VarType permits only variables of base type, not higher-order rules. Our approach is compatible with higher-order rules, as we show in Chapter 6, but for simplicity we leave them out of this Agda implementation. Contexts are lists of variable types, written with 'cons' on the right.

The types on the first line have their usual meaning. The type $\mathsf{D}^{+}\, D$ is the datatype named by D. Following Delphin (Poswolsky and Schürmann, 2008), we include a type $C\#$ classifying only the variables of type C. The type $\Psi \Rightarrow^{*} A$ of pre-derivabilities classifies inhabitants of

A in the current context extended with Ψ. The type $\Box A$ classifies closed inhabitants of A. The types $\forall_{\mathsf{c}}$ and $\exists_{\mathsf{c}}$ classify universal and existential context quantification; $\forall_{\leqslant C} A$ and $\exists_{\leqslant C} A$ provide bounded quantification over contexts containing only the type C.

Agda implementation

We now represent these types in Agda. We represent defined atoms, variable types and contexts as follows:

```
DefAtom = DefinedAtoms.Atom
data VarType : Set where
  ▷ : (D : DefAtom) {_ : Check (DefinedAtoms.world D)} → VarType
Vars = List VarType
```

DefinedAtoms.Atom is a parameter that we will instantiate later. DefinedAtoms.world returns true when D is allowed to appear in the context; Check turns this boolean into a proposition (Check True is the unit type; Check False is the empty type; see Chapter 2 for an introduction). A VarType is thus a pair of an atom along with the credentials allowing it to appear in contexts.

We represent the syntax of types in Agda as follows:

```
data Type : Set where
    -- types that have their usual meaning
  ‘1       : Type
  _⊗_      : Type → Type → Type
  ‘0       : Type
  _⊕_      : Type → Type → Type
  list_    : Type → Type
  _⊃_      : Type → Type → Type
    -- datatypes and context manipulation
  D⁺       : DefAtom → Type
  _#       : VarType → Type
  _⇒*_     : Vars → Type → Type
  □        : Type → Type
  ∀c       : (Vars → Type) → Type
  ∃c       : (Vars → Type) → Type
  ∀⩽       : VarType → Type → Type
  ∃⩽       : VarType → Type → Type
```

The only subtlety in this definition is that we represent the bodies of ∀c and ∃c by *admissibility functions* in Agda. This choice has some trade-offs: on the one hand, it means that the bodies of quantifiers can be specified by any Agda computation (e.g. by recursion over the domain). On the other hand, it makes it difficult to analyze the syntax of Types, because there is no way to inspect the body of the quantifier. Indeed, this caused problems for our implementation of the structural properties, which we solved by adding certain instances of the quantifiers (∀⇒ and ∃⇒, discussed below), which would otherwise be derived forms, as separate Type constructors.

In future work, we may pursue a more syntactic treatment of the quantifiers (which would of course be easier if we had good support for variable binding in Agda).

A *rule*, which is the type of a datatype constructor, pairs the defined atom being constructed with a single premise type (zero or many premises can be encoded using ‘1 and $\otimes$):

```
data Rule : Set where
  _⇐_ : DefAtom → Type → Rule
```

We will make use of a few derived forms:

- We write (∀⇒ A) for ∀c (λ Ψ → Ψ ⇒* A), and similarly for ∃⇒. This type quantifies over a context Ψ and immediately binds it around A. Similarly, we write [Ψ] * A for □ (Ψ ⇒* A)
- We write (C ⇒ Ψ) for ⇒* with a single premise.
- We write (C ⁺) for (D⁺ C) when C is a variable type.
- We write bool for ‘1 ⊕‘1 and A option for A ⊕‘1.

5.1.2 Semantics

In Figure 5.1, we define the semantic interpretation of a Type, a function <_>_ which maps each Type to a function from contexts to sets. We write an application of this function as < Ψ > A, where Ψ is a context and a A is a Type. The first six cases interpret the basic types of the simply-typed λ-calculus as their Agda counterparts, pushing the context inside to the recursive calls.

The next two cases interpret datatypes. We define an auxiliary datatype called Data which represents all of the data types defined in the universe. Data is indexed by a context and a defined atom, with the idea that the Agda set Data Ψ D represents the values of datatype D in context Ψ. There are two ways to construct a datatype: (1) apply a datatype constructor to an argument and (2) choose a variable from Ψ. Constants are declared in a signature, represented with a predicate on rules InΣ : Rule → Set, where InΣ R is inhabited when the rule R is in the signature. The first constructor, written as infix ·, pairs a constant with the interpretation of the constant's premise. The second constructor, ▷, injects a variable from Ψ into Data; the type of well-scoped de Bruijn indices _∈_ was discussed in Chapter 2. A DefAtom D is in the context if there exist credentials c for which the VarType formed by (▷ D {c}) is in the list Ψ.

Finally, we provide a collection of types that deal with the context: Ψ ⇒* A extends the context (we write + for append); □ A clears the context. The quantifiers ∀c and ∃c are interpreted as the corresponding Agda dependent function and pair types. Finally, the types ∀⩽ D A and ∃⩽ D A quantify over contexts Ψ′ for which AllEq Ψ′ D holds. The type AllEq says that every variable type in Ψ is equal to the given type D (List.all is true when its argument is true on all elements of the list; eqVarType is a boolean-valued equality function for variable types). (We could internalize AllEq Ψ′ D as a type alleq D—given meaning by < Ψ > (alleq D) = AllEq Ψ D—in which case the bounded quantifier could expressed as a derived form, but we have not needed alleq D in a positive position in the examples we have coded so far.)

An Agda datatype is *strictly positive* if it does not appear to the left of an Agda (admissibility) function type (→) in its own definition; this positivity condition ensures that the user does not

```
AllEq : Vars → VarType → Set
AllEq Ψ D = Check (ListM.all (eqVarType D) Ψ)
mutual
  data Data (Ψ : Vars) (D : DefAtom) : Set where
    _·_ : {A : Type} → InΣ (D ⇐ A) → < Ψ > A → Data Ψ D
    ▷ : {c : _} → (▷ D {c}) ∈ Ψ → Data Ψ D
  <_>_ : Vars → Type → Set
    -- basic types
  < Ψ >‘1          = Unit
  < Ψ >‘0          = Void
  < Ψ > (A ⊗ B) = (< Ψ > A) × (< Ψ > B)
  < Ψ > (A ⊕ B) = Either (< Ψ > A) (< Ψ > B)
  < Ψ > (list A) = List (< Ψ > A)
  < Ψ > (A ⊃ B) = (< Ψ > A) → (< Ψ > B)
    -- data types
  < Ψ > (D⁺ D)   = Data Ψ D
  < Ψ > (D #)    = D ∈ Ψ
    -- context manipulation
  < Ψ > (Ψnew ⇒* A) = < Ψ + Ψnew > A
  < _ > (□ A) = < [] > A
  < Ψ > (∃c τ) = Σ λ Ψ' → < Ψ > (τ Ψ')
  < Ψ > (∀c τ) = (Ψ' : Vars) → < Ψ > (τ Ψ')
  < Ψ > (∀⩽ D A) = (Ψ' : Vars) → AllEq Ψ' D → < Ψ + Ψ' > A
  < Ψ > (∃⩽ D A) = Σ (λ (Ψ' : Vars) → AllEq Ψ' D × < Ψ + Ψ' > A)
```

Figure 5.1: Interpretation of the universe

define general recursive types (e.g. $\mu D.D \to D$), which can be used to inhabit any type and to write non-terminating code. The above type Data does not pass the positivity checker: it is defined mutually with $<\_>\_$, and $<\_>\_$ occurs to the left of an Agda function type in the meaning of $\supset$. In this chapter, we wish to program with general recursive types, so we will ignore this failure of positivity checking. An interesting direction for future work would be to consider a total variant of our framework, which admits only strictly positive types. This would require a more refined explanation of the construction of the defined atoms in the universe, e.g. using containers (Abbott et al., 2005), because the positivity of a defined atom D depends on the rules for D in the signature InΣ.

We also define versions of $\Box$ and $\forall\Rightarrow$ that construct Agda Sets:

```
⊡ : Type → Set
⊡ A = < [] > A

∀⇛_ : Type → Set
∀⇛_ A = (Ψ : Vars) → < Ψ > A
```

The first is more concise than $< [] > \Box$ A, but means the same thing. The second is morally the same as $< [] > \forall\Rightarrow$ A, but this expands to $(\Psi : \mathsf{Vars}) \to < \Psi + [] >$ A, which is morally the same but intensionally different.

5.1.3 Structural Properties

In Figure 5.2, we present the type signatures for the structural properties; this is the interface that users of our framework see.

For example, the type of substitution should be read as follows: for any A and D, if the conditions for substitution hold, then there is a function of type $(\forall\Rightarrow (\mathsf{D} \Rightarrow \mathsf{A}) \supset (\mathsf{D}^{+}) \supset \mathsf{A})$ (for any context, given a term of type A with a free variable, and something of type D^{+} to plug in, there is a term of type A without the free variable). Weakening coerces a term of type A to a term with an extra free variable; strengthening does the reverse; exchange swaps two variables; contraction substitutes a variable for a variable. We also include an n-ary version of weakening for use with the bounded quantifier: if A can be weakened with D, then A can be weakened with a whole context comprised entirely of occurrences of D.

We discuss the meaning of the conditions (canSubst, etc.) below; in all of our examples, they will be discharged automatically by our implementation.

5.2 Examples

In this section, we illustrate programming in our framework, adapting a number of examples that have been considered in the literature (Pientka, 2008; Poswolsky and Schürmann, 2008; Shinwell et al., 2003). Throughout this section, we compare the examples coded in our framework with how they are/might be represented in Twelf, Delphin, Beluga, FreshML, and "raw Agda" (without using our framework). We endeavor to keep these comparisons objective, focusing on what invariants of the code are expressed, and what auxiliary functions the programmer needs

```
subst : (A : Type) {D : VarType}
  {_ : Check (canSubst (un▷ D) A)}
  → (∀⇛ (D ⇒ A) ⊃ (D ⁺) ⊃ A)
weaken : (A : Type) {D : VarType}
  {_ : Check (canWeaken (un▷ D) A)}
  → (∀⇛ A ⊃ (D ⇒ A))
strengthen : (A : Type) {D : VarType}
  {_ : Check (canStrengthen (un▷ D) A)}
  → ∀⇛ (D ⇒ A) ⊃ A
exchange2 : (A : Type) {D1 D2 : VarType}
  → (∀⇛ (D2 ⇒ D1 ⇒ A) ⊃ (D1 ⇒ D2 ⇒ A))
contract2 : (A : Type) {D : VarType}
  → (∀⇛ (D ⇒ D ⇒ A) ⊃ (D ⇒ A))
weaken*/bounded : (A : Type) (Ψ : Vars) {D : VarType}
  → (AllEq Ψ D)
  → {canw : Check (canWeaken (un▷ D) A)}
  → (∀⇛ A ⊃ (Ψ ⇒* A))
```

Figure 5.2: Type signatures of structural properties

to define. Several additional examples are available in the companion Agda code, including a translation from λ-terms to combinators, a type checker for simply-typed λ-calculus terms, an evaluator for λ-calculus with mutable references (using variables to represent locations), and an alternate version of normalization-by-evaluation, which has simpler types at the expense of slightly more-complicated code.

To use our framework, we give a type DefAtom representing the necessary datatypes names, along with a datatype

```
data In∑ : Rule → Set where
```

defining the datatype constructors.

We use the following naming convention: Defined atoms are given names that end in A. For types of variables, we define *atom*C to be *atom*A injected into VarType. Finally, we define *atom* to be the Type constructed by D⁺*atom*A. E.g. for a type of expressions exp, we will make the following definitions:

```
expA : DefAtom
expC = ▷ arithA
exp = D⁺ natA
```

5.2.1 Evaluating Arithmetic Expressions

As a simple example of mixing admissibility and derivability, we consider the syntax of a programming language with primitives specified by arbitrary meta-language functions. In particular, we consider arithmetic expressions constructed out of (1) variables, (2) numeric constants, (3) let binding, and (4) arbitrary binary primitive operations, represented by functions of type $\mathsf{nat} \supset \mathsf{nat} \supset \mathsf{nat}$.

In a concrete syntax, we might specify a signature for this datatype as follows:

$$\begin{array}{lcl} \mathsf{num} & : & \mathsf{arith} \Leftarrow \mathsf{nat} \\ \mathsf{letbind} & : & \mathsf{arith} \Leftarrow \mathsf{arith} \otimes (\mathsf{arith} \Rightarrow \mathsf{arith}) \\ \mathsf{binop} & : & \mathsf{arith} \Leftarrow \mathsf{arith} \otimes (\mathsf{nat} \supset \mathsf{nat} \supset \mathsf{nat}) \otimes \mathsf{arith} \end{array}$$

Recall that the symbol $\Leftarrow$ is used for datatype constructors, or rules. We use $\Rightarrow$ (derivability) to represent the body of the letbind, and $\supset$ (admissibility) to represent the primops. In Agda, this signature is rendered as constructors for the datatype $\mathsf{In}\Sigma$:

```
zero : InΣ (natA ⇐'1)
succ : InΣ (natA ⇐ nat)
num : InΣ (arithA ⇐ nat)
letbind : InΣ (arithA ⇐ arith ⊗ (arithC ⇒ arith))
binop : InΣ (arithA ⇐ arith ⊗ (nat ⊃ nat ⊃ nat) ⊗ arith)
```

Next, we define an evaluation function that reduces an expression to a number:

```
eval : ⊡ (arith ⊃ nat)
eval (num · n)             = n
eval (letbind · (e1, e2))  = eval (subst arith _ e2 e1)
eval (binop · (e1, f, e2)) = f (eval e1) (eval e2)
eval (▷ ())
```

Evaluation maps closed arithmetic expressions to natural numbers (the type $\boxdot\ (\mathsf{arith} \supset \mathsf{nat})$ reduces to the Agda function type `Data [] arithA → Data [] natA`). Constants evaluate to themselves; binops are evaluated by applying their code to the values of the arguments; let-binding is evaluated by substituting the expression e1 into the letbind's body e2[1] and then evaluating the result. A simple variation would be to evaluate e1 first and then substitute its value into e2. The final clause covers the case for variables with a refutation pattern: there are no variables in the empty context.

[1]The arith argument to subst is the type A in the D ⇒ A argument to substitution; Agda's type reconstruction procedure requires this annotation. The underscore is the context argument instantiating the ∀⇛ in the type of subst; this could be eliminated by adding an implicit context quantifier (whose meaning is {Ψ : Vars} → ...) to the universe. The credentials for performing substitution are marked as an implicit argument, so there is no evidence of it visible in the call to subst.

Comparison. This example provides a nice illustration of the benefits of our approach: Substitution is provided "for free" by the framework, which infers that it is permissible to substitute for arithC variables in arith—this improves on implementing the example in raw Agda, where substitution would need to be implemented by hand. The type system enforces the invariant that evaluation produces a closed natural number.

It is not possible to define the type arith in Twelf/Delphin/Beluga, as LF representations cannot use computational functions. One could program this example in FreshML, but it would be necessary to implement substitution directly for arith, as FreshML does not provide a generic substitution operation.

Agda checks that eval's pattern matching is exhaustive. However, Agda is not able to verify the termination of this function (even setting aside the positivity problems—which are not really problematic here, as nat is morally defined prior to arith, so the definition is iterated), as it recurs on a substitution-instance of one of the inputs. Setting aside the computational functions in binop, it would be possible to get the call-by-value version of this code to pass Twelf's termination checker, which recognizes certain substitution instances as smaller. We have not yet investigated how to explain this induction principle to Agda.

5.2.2 Reduction

Next, we implement small-step reduction for the λ-calculus. This example illustrates recursion over open terms, as well as additional uses of the structural properties provided by our framework. We represent the λ-calculus with the following signature, which is familiar from higher-order abstract syntax:

$$\mathsf{lam} \;:\; \mathsf{In}\Sigma\ (\mathsf{expA} \Leftarrow \mathsf{expC} \Rightarrow \mathsf{exp})$$
$$\mathsf{app} \;:\; \mathsf{In}\Sigma\ (\mathsf{expA} \Leftarrow \mathsf{exp} \otimes \mathsf{exp})$$

Next, we define a function red implementing small-step reduction. As a first cut, we might try assigning red the type $\boxdot\ (\mathsf{exp} \supset \mathsf{exp} \oplus \text{`}1)$, which is interpreted as $\mathsf{Data}\ [\,]\ \mathsf{exp} \to \mathsf{Either}\ (\mathsf{Data}\ [\,]\ \mathsf{exp})\ \mathsf{Unit}$. That is, red e returns an option, with Inl e′ signaling that e steps to e′, and Inr <> signaling that e cannot be reduced further. However, because reduction proceeds under binders, it is necessary to generalize red so that it works on open terms. Thus, we instead give it the type

$$\mathsf{red} \;:\; \forall\Rrightarrow\ \mathsf{exp} \supset \mathsf{exp} \oplus \text{`}1$$

which works in any context. This type is interpreted as

$$(\Psi \;:\; \mathsf{Ctx}) \;\to\; \mathsf{Data}\ \Psi\ \mathsf{exp} \;\to\; \mathsf{Either}\ (\mathsf{Data}\ \Psi\ \mathsf{exp})\ \mathsf{Unit}$$

red is implemented as follows:

```
red Ψ (▷ x)  =  Inr <>
red Ψ (app · (lam · e, e2))  =  Inl (subst exp Ψ e e2)
red Ψ (app · (e1, e2)) with red Ψ e1  |  red Ψ e2
...                    |    Inl e1'    |  _         =  Inl (app · (e1', e2))
...                    |    Inr <>     |  Inl e2'   =  Inl (app · (e1, e2'))
...                    |    Inr <>     |  Inr <>    =  Inr <>
red Ψ (lam · e) with red (expC :: Ψ) e
...                |    Inl e'  =  Inl (lam · e')
...                |    Inr <>  =  Inr <>
```

The variable case (▷ x) says that variables do not reduce. The second line reduces a β-redex using substitution; the fact that subst is available for exp is inferred by our framework. For any other application, we try reducing the function position. If this succeeds, we have a reduction; otherwise, we try reducing the argument position. For a lam, we try reducing the body, which is an example of making a recursive call in an extended context expC :: Ψ, which extends Ψ with an additional assumption of type exp.

Comparison. If a programmer were to implement this example in raw Agda, the signature would be more verbose, as it would need to mention contexts explicitly, and he would have to implement subst by hand. In Twelf, reduction would be written as a logic program, so the failure cases would not need to be described explicitly. In Twelf and Delphin, the context is managed implicitly, so the Ψ arguments would be elided. In Beluga, the code would look quite similar to the above.

5.2.3 Type checking

Next, we implement a type checker for the simply-typed λ-calculus. The function red above worked for any context Ψ, because the variable case simply returned a constant. In contrast, to implement a type checker, we need to maintain extra information about the context, associating a type with each term variable. To maintain this information, we use a function whose domain is the variables of a given type, a technique inspired by Delphin (Poswolsky and Schürmann, 2008).

We represent the STLC with the following signature:

```
arr  :  InΣ (tpA ⇐ tp ⊗ tp)
unit :  InΣ (tpA ⇐'1)
fn   :  InΣ (tmA ⇐ □ tp ⊗ (tmC ⇒ tm))
ap   :  InΣ (tmA ⇐ tm ⊗ tm)
mt   :  InΣ (tmA ⇐'1)
```

For simplicity, we make the fact that all types are closed explicit, using a □ in the type of the constant fn to say that the type annotation is closed. This saves us from having to weaken and strengthen term variables in types (see size below).

To implement the type checker, we first need equality of types, which is implemented by a simple recursion over closed types:

```
eqtp : ⊡ (tp ⊃ tp ⊃ bool)
```

Next, we define an environment to be a function that gives a closed type for each term variable; recall that the type tmC # means the variables of type tm:

```
env : Type
env = tmC # ⊃ □ tp
```

Type synthesis takes a term and an environment and optionally returns a closed type:

```
synth : (∀⇛ tm ⊃ env ⊃ (□ tp) ⊕'1)
```

It is implemented as follows:

```
synth Ψ (▷ x) env = Inl (env x)
synth Ψ (fn · (A, e)) env with synth (tmC :: Ψ) e (extend {□ tp} {tmC} Ψ env A)
...                          |    Inl B = Inl (arr · (A, B))
...                          |    Inr <> = Inr <>
synth Ψ (ap · (e1, e2)) env with synth Ψ e1 env     | synth Ψ e2 env
...                           |    Inl (arr · (A1, A2)) | Inl B1 with eqtp A1 B1
...                                                     |    Inl <> = Inl A2
...                                                     |    Inr <> = Inr <>
synth Ψ (ap · (e1, e2)) env |    _                    | _ = Inr <>
synth Ψ (mt · <>) env = Inl (unit · _)
```

A variable has the type given in the environment; the framework already ensures well-scopedness, so there is no possibility of getting a variable out of scope. The clause for ap check the function and argument positions and then check that the domain of the function is the type of the argument. The case for fn is interesting because it must extend the environment env by mapping the last variable mapped to the type A. We do this using the function extend:

```
extend : ∀ {A D} → (∀⇛ (D # ⊃ A) ⊃ A ⊃ (D ⇒ D #) ⊃ A)
extend Ψ σ new i0 = new
extend Ψ σ new (iS i) = σ i
```

If we had made a mistake and forgotten to extend the environment, the Agda type checker would helpfully complain that (tmC :: Ψ) != Ψ.

Comparison. We have included this example mainly to introduce the environment technique; it would not be very different in raw Agda, as we have not made use of any of the structural properties. In Twelf, the failure cases could be elided, unification could be used for type equality, and the LF context could be used to represent the environment implicitly. In Delphin, the context Ψ would be implicit, but the environment would be passed explicitly. In Beluga, the LF context could be used to associate variables with types, but it is passed explicitly.

5.2.4 Closure-based Evaluator

Next, we implement a closure-based evaluator for the untyped λ-calculus. Like the type checker above, this example illustrates the use of functions to maintain invariants about the context; additionally, it shows how to use existential quantification over contexts. We gave the signature for λ-terms above; closures are represented by a type clos as follows:

```
closure : In Σ (closA ⇐ (∃⇒ (expC ⇒ exp) ⊗ (expC # ⊃ □ clos)))
```

The type of closures, clos, is a recursive type with one constructor closure. The premise of closure should be read as follows: a closure is constructed from a triple $(\Psi, \mathsf{e}, \sigma)$, where (1) Ψ is an existentially quantified context; (2) e is an expression in Ψ with an extra free variable, which represents the body of a λ-abstraction; and (3) σ is a substitution of closed closures for all the variables in Ψ. We represent a substitution as a function that maps each expression variable in the context (classified by the type expC #) to a closure. The type of the premise provides a succinct description of all of this: $\exists\Rightarrow$ introduces the variables in the existentially quantified context into scope without explicitly naming the context; $\Rightarrow$ extends the context with an additional variable; (expC #) ranges over all of the variables in scope. For comparison, in Ψ this type reduces to the Agda type

```
Σ (λ (Ψ' : Vars) → (Data (Ψ + Ψ', , expC) expA) × (expC ∈ (Ψ + Ψ') → Data [] closA))
```

(where we write , , for cons on the right).

As in the type checker example we recur over open expressions, so eval is quantified over an unknown context Ψ using $\forall\Rightarrow$. Evaluation takes two further arguments: (1) an expression with free variables in Ψ, and (2) an environment, represented by a function that yields a closed closure for each expression variable in Ψ; eval returns a closed closure.

```
env : Type
env = expC # ⊃ □ clos
eval : ⊡ (∀⇒ exp ⊃ env ⊃ □ clos)
eval Ψ (▷ x) σ = σ x
eval Ψ (lam · e) σ = closure · (Ψ, e, σ)
eval Ψ (app · (e1, e2)) σ
  with eval Ψ e1 σ
... |   closure · (Ψ', e', σ') = eval (Ψ', , expC) e' (extend {(□ clos)} _ σ' (eval Ψ e2 σ))
... |   ▷ x = impossible x
```

A variable is evaluated by applying the substitution. A lam evaluates to the obvious closure. To evaluate an application, we first evaluate the function position. Case-analyzing the evaluation of e1 gives two cases: (1) the value is constructed by the constructor closure; (2) the value is a variable.

In the first case, we evaluate the body of the closure in an extended environment. The call to the function extend, described in the type checker example above, extends the environment σ' so that the last variable is mapped to the value of e2. At the call site of extend, we must explicitly

supply the type A (in this case □ clos) to help out type reconstruction. The underscore stands for the instantiation of the ∀⇒, which is marked as an explicit argument, but can in this case be inferred.

The second case is contradicted using the function impossible, which refutes the existence of a variable at a non-VarType—which clos is, because we never wish to have clos variables.

The context argument Ψ to eval does not play an interesting role in the code, but Agda's type reconstruction requires us to supply it explicitly at each recursive call. Agda is unable to verify the termination of this evaluator for the untyped λ-calculus, as one would hope.

When writing this code, one mistake a programmer might make is to evaluate the body of the closure in σ instead of σ', which would give dynamic scope instead of static scope. If we make this mistake, Agda highlights the occurrence of σ and helpfully reports the type error that Ψ′ != Ψ, indicating that the context of the expression does not match the context of the substitution.

Comparison. In Twelf, one cannot represent substitutions σ using admissibility functions, because these are not available for use in LF encodings. However, because the domain of the substitution is finite, a first-order representation of substitutions could be used. Additionally, Twelf does not provide the □ and ∃⇒ connectives that we use here to describe the contexts of closures. While it should be possible for the programmer to express the necessary context invariants using explicit contexts (Crary, 2008), this is a fairly heavy encoding technique. Because of these two limitations, the resulting Twelf code would be more complicated than the above. One would hope for better Delphin and Beluga implementations than a port of the Twelf code, but Delphin lacks existential context quantification and □, and Beluga lacks the parameter type exp #, so our definition of clos cannot be straightforwardly ported to either of these languages. Beluga provides a built-in type of substitutions, written [Ψ']Ψ, so one might hope to represent closures as $\exists\psi.([\psi, x : \mathsf{exp}]\mathsf{exp}) \times [.]\psi$; however, the second component of this pair associates an *expression* with each expression variable in ψ, whereas, in this example, we need to associate a *closure* with each expression variable in ψ. One could implement this example in FreshML (Shinwell et al., 2003), but the type system would not enforce the invariant that closures are in fact closed. To our knowledge, a proof of this property for this example has not been attempted in Pure FreshML (Pottier, 2007), though we know of no reason why it would not be possible. The only improvement over raw Agda in this case is that the types are more concise when specified in our point-free notation.

5.2.5 Variable Manipulation

Next, we consider some common examples that involve manipulation of variables. None of these examples are very different than what one would write in raw Agda; instead, we show them to illustrate how dependent de Bruijn indices compare with other representations of variables.

Size

First, we compute the size of a λ-term. Addition is defined as usual, with a contradictory variable case because no natA variables are allowed.

```
plus : ⊡ (nat ⊃ nat ⊃ nat)
plus (zero · _) m = m
plus (succ · n) m = succ · (plus n m)
plus (▷ ()) _

size : (∀⇛ exp ⊃ □ nat)
size Ψ (▷ x)             = succ · (zero · _)
size Ψ (app · (e1, e2)) = succ · (plus (size Ψ e1) (size Ψ e2))
size Ψ (lam · e)         = succ · (size (Ψ, , expC) e)
```

Agda successfully termination-checks these functions.

The type of size expresses that it returns a closed natural number. For comparison, we implement a second version that does not make this invariant explicit:

```
size' : ⊡ (∀⩽ expC (exp ⊃ nat))
size' Ψ bound (▷ x) = succ · (zero · _)
size' Ψ bound (app · (e1, e2)) =
  succ · (plus' Ψ bound (size' Ψ bound e1) (size' Ψ bound e2)) where
    plus' : ⊡ (∀⩽ expC (nat ⊃ nat ⊃ nat))
    plus' Ψ b = weaken*/bounded (nat ⊃ nat ⊃ nat) Ψ {expC} b [] plus
size' Ψ bound (lam · e) = strengthen nat _ (size' (Ψ, , expC) bound e)
```

Without the □, size must return a number in context Ψ: in the application case, we must weaken plus into Ψ, and in the lam case we must strengthen the extra expC variable out of the recursive call. Strengthening expression variables from natural numbers is permitted by our implementation of the structural properties because natural numbers cannot mention expressions; we use a subordination-like analysis to determine this (Virga, 1999). To ensure that these weakenings and strengthenings are permitted, we type size' with a bounded quantifier over exp.

Comparison. The first version is similar to what one writes in FreshML, except in that setting there is no need to pass around a context Ψ. In the second version, the strengthening of the recursive result in the lam case is analogous to the need, in FreshML 2000 (Pitts and Gabbay, 2000), to observe that nat is pure (always has empty support); FreshML (Shinwell et al., 2003) does not require this.

In Beluga, one can express either the first or second versions. In Twelf and Delphin, one can only express the second variation, as these languages do not provide □. However, the Twelf/Delphin/Beluga syntax for weakening and strengthening is terser than what we have been able to construct in Agda: weakening is not marked in the proof term; strengthening is marked by pattern-matching the result of the recursive call and marking those variables that *do* occur, which in this case does not include the expression variable. For example, the lam case of size in Twelf looks like this:

```
- : size (lam ([x] E x)) (succ N)
  ← ({x : exp} size (E x) N) ∘
```

Twelf's coverage checker verifies that expression variables can be strengthened out of natural numbers when checking this case.

Counting occurrences of a variable

A simple variation is to count the number of occurrences of a distinguished free variable. The input to this function has type $(\mathsf{expC} \Rightarrow \mathsf{exp})$, and we count the occurrences of the bound variable:

```
cnt : ∀⇛ (expC ⇒ exp) ⊃ □ nat
cnt Ψ (▷ i0)           = succ · (zero · _)
cnt Ψ (▷ (iS _))       = zero · _
cnt Ψ (app · (e1, e2)) = plus (cnt Ψ e1) (cnt Ψ e2)
cnt Ψ (lam · e)        = cnt (Ψ, , expC) (exchange2 exp Ψ e)
```

In the first two cases, we pattern-match on the variable: when it is the last variable, the last variable occurs once; when it is not, it occurs zero times. The lam case recurs on the exchange of e, so that the last variable remains the one we are looking for. Agda fails to termination-check this example because it recurs on the result of exchange.

Comparison. Pattern-matching on variables is represented using higher-order metavariables in Twelf/Delphin/Beluga and using equality tests on names in FreshML. The exchange needed in the lam case is written as a substitution in the Twelf/Delphin/Beluga version of this clause. In Twelf one would write:

```
- : cnt ([x] lam ([y] E x y)) N
  ← ({y:exp} cnt ([x] E x y) N) ∘
```

In the input to this clause, the metavariable E, which stands for the body of the function, refers to the last variable in the context (the lam-bound variable) as y and the second-last variable (the variable being counted) as x. In the recursive call, y is exchanged past the binding of x, so the instantiation E x y swaps "last" and "second-last".

Computing free variables

Next, we consider a function computing the free variables of an expression. This function has the following type: $(\forall\Rrightarrow \mathsf{exp} \supset \mathsf{list}\ (\mathsf{expC}\ \#))$—in any context, this function accepts an expression in that context and produces a list of variables in that context. This typing ensures that we do not accidentally return a bound variable.

```
remove : {D : VarType}
  → (∀⇛ (D ⇒ list (D #)) ⊃ list (D #))
remove Ψ []          = []
remove Ψ (i0 :: ns)  = (remove Ψ ns)
remove Ψ ((iS i) :: ns) = i :: (remove Ψ ns)

fvs : (∀⇛ exp ⊃ list (expC #))
fvs Ψ (▷ x)          = [x]
fvs Ψ (lam · e)      = remove Ψ (fvs (Ψ, , expC) e)
fvs Ψ (app · (e1, e2)) = append (fvs Ψ e1) (fvs Ψ e2)
```

In the lam case, we use the helper function remove to remove the lam-bound variable from the recursive result. The function remove takes a list of variables, itself with a distinguished free

variable, and produces a list of variables without the distinguished variable. If the programmer were to make a mistake in the second clause by accidentally including i0 in the result, he would get a type error. Agda successfully termination-checks this example.

Comparison. For comparison with FreshML (Shinwell et al., 2003), the type given to remove here is analogous to their Figure 6:

```
remove : (<name> (name list)) → name list
```

where <a>τ is a nominal abstractor. The authors comment that they prefer the version of remove in their Figure 5:

```
remove : name → (name list) → name list
```

where the name to removed is specified by the first argument, rather than using a binder.

Using dependent types, we can type this second version of remove as follows:

```
remove : (Ψ : Vars) (i : exp ∈ Ψ)
   → List (exp ∈ Ψ) → List (exp ∈ (Ψ - i))
```

where Ψ - i removes the indicated element element from the list. This type is of course expressible in Agda, but not in the simply-typed universe that we consider in this chapter.

In Twelf, this example can be coded with two differences from the above. The first is that remove pattern-matches on derivability functions, rather than pattern-matching to distinguish different variables in the (extended) context, as we do above:

```
remove : (var → list) → list → type.
- : remove ([x] cons x (L x)) L'
   ← remove L L'.
- : remove ([x] cons Y (L x)) (cons Y L')
   ← remove L L'.
- : remove ([x] nil) nil.
```

Second, to enforce the invariant that the output list contains only variables, it is necessary to modify the encoding to identify variables. One option is to use a separate type var for variables, as we have done here (though this necessitates a manual proof that an expression can be substituted for a variable); another is to define a predicate isvar E which holds when E is a variable. Without this invariant, remove is not total, though it could be extended to a total relation by adding cases for the remaining expressions.

In Delphin, one would need to match on a derivability function, as Delphin does not support matching against individual free variables, but the # type could be used to define a list of variables (if Delphin were extended with non-LF inductive types). Complementarily, in Beluga, one can match against individual variables by restricting a pattern by a substitution, but the encoding would need to explicitly provide a type of variables.

η-Contraction

In Twelf/Delphin/Beluga, one can recognize η-redices by writing a meta-variable that is not applied to all enclosing locally bound variables. E.g. in Twelf one would write

```
- : contract (lam [x] app F x) F.
```

The metavariable F:exp is bound outside the scope of x, and thus stands only for terms that do not mention x. (To allow it to mention x, we would bind F:exp $\rightarrow$ exp and write (F x) in place of F.)

Unfortunately, Agda does not provide this sort of pattern matching for our encoding—pattern variables are always in the scope of all enclosing local binders—so we must explicitly call a strengthening function that checks whether the variable occurs:

```
strengthen? : ∀⇛ (expC ⇒ exp) ⊃ exp option
strengthen? Ψ (▷ i0)      = Inr _
strengthen? Ψ (▷ (iS i)) = Inl (▷ i)
strengthen? Ψ (app · (e1, e2)) with strengthen? Ψ e1 | strengthen? Ψ e2
...                              | Inl e1'          | Inl e2' = Inl (app · (e1', e2'))
...                              | _               | _       = Inr _
strengthen? Ψ (lam · e) with strengthen? (Ψ, , expC) (exchange2 exp Ψ e)
...                          |    Inl e' = Inl (lam · e')
...                          |    _     = Inr _
contract-η : ∀⇛ exp ⊃ exp option
contract-η Ψ (lam · (app · (f, ▷ i0))) = strengthen? Ψ f
contract-η Ψ _ = Inr <>
```

We conjecture that strengthen? could be implemented datatype-generically for all purely positive types (no $\supset$ or $\forall$c or $\forall\leqslant$)—it is not possible to decide whether a variable occurs in the values of these function types (cf. FreshML, where it is not possible to decide whether a name is in the support of a function). This strengthening function is not an instance of the generic map that we define below, as it changes the type of the term (exp to exp option); though there may be a more general traversal that admits this operation.

5.2.6 Combinators

Another place where the strengthening issue discussed with size above comes up is in translations between languages. A standard example is translation from λ-terms to combinators, which we represent as follows:

```
comb/s   : InΣ (combA ⇐'1)
comb/k   : InΣ (combA ⇐'1)
comb/i   : InΣ (combA ⇐'1)
comb/app : InΣ (combA ⇐ comb ⊗ comb)
```

First, we define bracket-abstraction, which takes a combinator with a free variable, and abstracts over it, using combinators to pass the value of the variable to its use sites:

```
bracket : ∀⇛ ((combC ⇒ comb) ⊃ comb)
bracket Ψ (▷ i0) = comb/i · <>
bracket Ψ (▷ (iS x)) = comb/app · (comb/k · <>, ▷ x)
bracket Ψ (comb/i · <>) = comb/app · (comb/k · <>, comb/i · <>)
bracket Ψ (comb/s · <>) = comb/app · (comb/k · <>, comb/s · <>)
bracket Ψ (comb/k · <>) = comb/app · (comb/k · <>, comb/k · <>)
bracket Ψ (comb/app · (a1, a2)) =
  comb/app · (comb/app · (comb/s · <>, bracket Ψ a1), bracket Ψ a2)
```

If the combinator is the distinguished last variable (i0), then the result is the I combinator. If it is any other variable, the result is the constant K combinator applied to that variable. Similarly, the result for any constant among I, S, and K is K applied to that constant. Application bracket-abstracts the two pieces and combines them with S. The type of bracket ensures that the variable being abstracted is not free in the result.

The translation from expressions to combinators maintains an environment mapping each expression variable to a combinator variable:

```
combw = (expC # ⊃ combC #)
extend-combenv : ∀⇛ combw ⊃ (expC ⇒ combC ⇒ combw)
extend-combenv Ψ σ (iS i0) = i0
extend-combenv Ψ σ (iS (iS i)) = iS (iS (σ i))
```

The function extend-combenv shows how to extend an environment in Ψ to an environment in $\Psi, , \mathsf{exp}, , \mathsf{comb}$, where the new expression variable is mapped to the new combinator variable, and all other results are shifted so that they make sense in the new environment.

The translation is defined as follows:

```
tr : ∀⇛ (combw ⊃ exp ⊃ comb)
tr Ψ σ (▷ x)             = ▷ (σ x)
tr Ψ σ (app · (e1, e2)) = comb/app · (tr Ψ σ e1, tr Ψ σ e2)
tr Ψ σ (lam · e)         =
  let
    r : < Ψ, , expC, , combC > comb
    r = tr (Ψ, , expC, , combC) (extend-combenv Ψ σ) (weaken exp (Ψ, , expC) e)
    s : < Ψ, , combC > comb
    s = strengthen/anywhere comb (iS i0) r
  in
    bracket Ψ s
```

The interesting case is lam, where we first let r stand for the result of a recursive call in an extended context. Prior to the recursive call, we must weaken e, which is in context $\Psi, , \mathsf{expC}$, with the new combinator variable. r is then a combinator in context $\Psi, , \mathsf{expC}, , \mathsf{combC}$. Next, we

can strengthen away the expC assumption, as expressions cannot occur in combinators. Finally, bracket abstraction gives the result.

An alternative is to use different contexts for the expression and the combinator term, in which case no weakening and strengthening are necessary:

```
combw' : Vars → Vars → Type
combw' Ψ Ψ' = ([Ψ] * expC #) ⊃ ([Ψ'] * combC #)
tr' : ⊡ (∀c (λ Ψ → ∀c (λ Ψ' → combw' Ψ Ψ' ⊃ ([Ψ] * exp) ⊃ ([Ψ'] * comb))))
tr' Ψ Ψ' σ (▷ x)            = ▷ (σ x)
tr' Ψ Ψ' σ (app · (e1, e2)) = comb/app · (tr' Ψ Ψ' σ e1, tr' Ψ Ψ' σ e2)
tr' Ψ Ψ' σ (lam · e)        = bracket Ψ' (tr' (Ψ, , expC) (Ψ', , combC) newσ e) where
  newσ : ⊡ (combw' (Ψ, , expC) (Ψ', , combC))
  newσ i0 = i0
  newσ (iS i) = (iS (σ i))
```

The price is that the type is more verbose, because the context can no longer be treated implicitly.

Comparison. An implementation in both Twelf and Delphin is more similar to tr than tr', in that both the expression and the combinator term exist in the same ambient LF context. When writing this code in Twelf, the LF context can be used to represent the environment, weakening is tacit, and strengthening is represented by pattern-matching, as discussed above. In Delphin, the environment is represented as a function, as we have done here, but weakening and strengthening are treated as in Twelf. In Beluga, one can implement the function in the tr style, but the environment is represented in the LF context, and weakening and strengthening are notated using substitutions, as in Twelf. One can almost write the type of the tr'-style implementation, but it is unclear how to represent the environment (Pientka (2006) suggests using the tr style instead). Relative to raw Agda, the type of tr is more concise because the context does not need to be mentioned explicitly; tr' is no different.

5.2.7 Normalization by Evaluation

In Figure 5.3, we present an interesting example that mixes derivability and admissibility, β-normalization-by-evaluation for the untyped λ-calculus. Normalization-by-evaluation (NBE) is a method for evaluating λ-terms using the evaluator of the meta-language, rather than defining normalization syntactically, using substitution. The idea of NBE is to interpret the syntax of an object language into the meta-language, interpreting derivability functions in the syntax as admissibility functions in the meta-language. The obvious way to do this is to interpret the object-language type A arrow B as the meta-language type ⟦ A ⟧ → ⟦ B ⟧. However, with this interpretation, it is impossible to read off a syntactic normal form, because functions are interpreted as black-box admissibility functions. The solution is to interpret into a space of meta-language functions that can be applied to syntactic, as well as semantic, arguments. A syntactic normal form can then be read off by applying such a function to a variable.

For the untyped λ-calculus, our semantic domain sem consists of syntactic *neutral* terms of the form x e1 ... en as well as admissibility functions from sem ⊃ sem. However, in implementing NBE, it is necessary to weaken values of type sem into bigger contexts, and the type

```
napp : In Σ (neuA ⇐ neu ⊗ sem)
neut : In Σ (semA ⇐ neu)
slam : In Σ (semA ⇐ (∀⩽ neuC (sem ⊃ sem)))
```

```
reifyenv : Vars → Vars → Type
reifyenv Ψe Ψs = ([Ψs] * (neuC #) ⊃ [Ψe] * (expC #))
reifyn : ∀C (λ Ψe → ∀c (λ Ψs → reifyenv Ψe Ψs ⊃ [Ψs] * neu ⊃ [Ψe] * exp))
reifyn Ψe Ψs σ (▷ x) = ▷ (σ x)
reifyn Ψe Ψs σ (napp · (n, s)) =
  app · (reifyn Ψe Ψs σ n, reify Ψe Ψs σ s)
reify : ∀C (λ Ψe → (∀c λ Ψs → reifyenv Ψe Ψs ⊃ [Ψs] * sem ⊃ [Ψe] * exp))
reify Ψe Ψs σ (slam · φ) =
  lam · reify (Ψe, , expC) (Ψs, , neuC) σ' (φ [neuC] _ (neut · (▷ i0))) where
    σ' : ⊡ (reifyenv (Ψe, , expC) (Ψs, , neuC))
    σ' i0 = i0
    σ' (iS x) = iS (σ x)
reify Ψe Ψs σ (neut · n) = reifyn Ψe Ψs σ n
reify Ψe Ψs σ (▷ x) = impossible x
```

```
appsem : ∀⇛ sem ⊃ sem ⊃ sem
appsem _ (slam · φ) s2 = φ [] _ s2
appsem _ (neut · n) s2 = neut · (napp · (n, s2))
appsem _ (▷ x) _ = impossible x
evalenv : Vars → Vars → Type
evalenv Ψe Ψs = [Ψe] * (expC #) ⊃ ([Ψs] * sem)
eval : ∀C (λ Ψe → ∀c (\Ψs → evalenv Ψe Ψs ⊃ [Ψe] * exp ⊃ [Ψs] * sem))
eval Ψe Ψs σ (▷ x) = σ x
eval Ψe Ψs σ (app · (e1, e2)) = appsem Ψs (eval Ψe Ψs σ e1) (eval Ψe Ψs σ e2)
eval Ψe Ψs σ (lam · e) = slam · φ where
  φ : < Ψs > (∀⩽ neuC (sem ⊃ sem))
  φ Ψ' ctxinv s' = eval (Ψe, , expC) (Ψs + Ψ') σ' e where
    σ' : < [] > evalenv (Ψe, , expC) (Ψs + Ψ')
    σ' i0 = s'
    σ' (iS i) = weaken*/bounded sem Ψ' {neuC} ctxinv Ψs (σ i)
```

Figure 5.3: Normalization by evaluation

sem ⊃ sem is not weakenable with variables of type sem. The solution is to consider admissibility functions that explicitly quantify over all future extensions of the context, which we represent with the type (∀⇒ sem ⊃ sem). This is similar to a Kripke semantics, or an interpretation into presheaves. However, for the argument to go through, we must ensure that the context extension Ψ' consists only of variables of a specific type neu, so we use a bounded context quantifier below.

We represent the semantics by the datatypes neu and sem in Figure 5.3.[2] The type neu (neutral terms) consists of variables or neutral terms applied to semantic arguments (napp); these are the standard neutral proofs in natural deduction. A sem (semantic term) is either a neutral term or a semantic function. A semantic function of type (∀⩽ neuC (sem ⊃ sem)) is an admissibility function that works in any extension of the context consisting entirely of neutral variables.

We define reification first, via two mutually recursive functions, reifyn (for neutral terms) and reify (for semantic terms). It is typical in logical relations arguments to use two independent contexts, one for the syntax and one for the semantics. Thus, we parametrize these functions by two contexts, one consisting for neu variables for the semantics, and the other consisting of exp variables for the syntax. We will write Ψs for the former and Ψe for the latter. The type of reify then says that, given an environment mapping neutral variables in Ψs to expression variables in Ψe, reify maps semantics in the semantic context to expressions in the expression context

Even though reify is given a precise type describing the scoping of variables, its code is as simple as one could want. To reify neutral terms: The reification of a variable is the variable given in the substitution. The reification of an application is the application of the reifications. To reify semantic terms: The reification of a function (slam · φ) is the λ-abstraction of the reification of an instance of φ. In the recursive call, the expression context is extended with a new exp variable (which is bound by the lam) and the semantic context is extended with a new neu variable. We instantiate the semantic function φ, which anticipates extensions of the context, with this one-variable extension ([x] constructs a singleton list), and apply it to the variable. σ' makes the "parallel" extension of σ, mapping the one new variable to the other. This could be abstracted into a library function if one wished. The neutral-to-semantic coercion is reified recursively, and we disallow sem variables from the context.

To define evaluation, we first define an auxiliary function appsem that applies one semantic term to another. This requires a case-analysis of the function term: when it is an slam (i.e. the application is a β-redex), we apply the embedded computational function, choosing the nil context extension, and letting the argument be s2. When the function term is neutral, we make a longer neutral term.

The type of eval is symmetric to reify, except the environment that we carry along in the induction maps expression variables to semantic *terms* rather than just variables. A variable is evaluated by looking it up; an application is evaluated by combining the recursive results with semantic application. A lam is evaluated to an slam whose body φ has the type indicated in the figure. When given a context extension Ψ' and an argument s' in that extension, φ evaluates the original body e in an extended substitution. The new substitution σ' maps the λ-bound variable

[2] In a previous account of this work (Licata and Harper, 2009), we wrote the types in this example in a more concise, but also less intuitive, way: One can choose to use the ambient context as either the syntactic or semantic context, rather than mentioning both explicitly. However, the choice is arbitrary, and the asymmetry makes the code hard to read. Additionally, one can abstract out the definitions of the environment types and environment extension, but we inline these definitions here for simplicity.

i0 to the provided semantic value, and defers to σ on all other variables. However, σ provides values in Ψs, which must be weakened into the extension Ψ'. Fortunately, the bounded quantifier provides sufficient evidence to show that weakening can be performed in this case, because sem's can be weakened with neu variables.

Normalization is defined by composing evaluation and reification. We define a normalizer for closed λ-terms as follows:

```
emptyreifyenv : ⊡ (reifyenv [] [])
emptyreifyenv ()

emptyevalenv : ⊡ (evalenv [] [])
emptyevalenv ()

norm : ⊡ (exp ⊃ exp)
norm e = reify [] [] emptyreifyenv (eval [] [] emptyevalenv e)
```

Our type system has verified the scope-correctness of this code, proving that it maps closed terms to closed terms. Amusingly, Agda accepts the termination of this evaluator for the untyped λ-calculus, provided that we have told it to ignore its issues with our universe itself—a nice illustration of the need for the positivity check on datatypes.

As with the combinator example, it is possible to give an alternate version of this code where both expressions and semantic terms live in the same context, at the expense of various appeals to weakening and strengthening are necessary. For comparison, we include this version in Figure 5.4. As with the combinators example, the binder cases require weakening before the recursive call, and strengthening afterwards. Additionally, the substitution extension σ' in eval is more complicated, as it must handle the potential additional exp variables in Ψ'—which do not in fact exist, as Ψ' is made up of only neu variables.

Comparison. The type sem mixes derivability and admissibility in an essential way: it must use admissibility to inherit evaluation from Agda, and it must use derivability to make it possible to read off a normal form. The premise $(\forall\leqslant \mathsf{neuC}\ (\mathsf{sem} \supset \mathsf{sem}))$ uses both $\Rightarrow$ and $\supset$ (recall that there is a $\Rightarrow$ buried in the definition of $\forall\leqslant$). Because it uses $\supset$ in a recursive datatype, it is not representable in LF. Because it uses $\Rightarrow$, it would not even be representable in Delphin/Beluga extended with standard recursive types (that did not interact with the LF part of the language). Despite the fact that our implementation enforces strong invariants about the scope of variables, the code is essentially as simple as the FreshML version described by Shinwell et al. (2003), aside from the need to pass the contexts Ψe and Ψs along. Invariants about variable scoping can be proved in Pure FreshML (Pottier, 2007), but we enforce these invariants within a type system, not using an external specification logic. Relative to a direct implementation in Agda (see Pouillard and Pottier (2010)) our framework provides the weakening function needed in the final case of eval for free.

5.3 Structural Properties

Next, we describe our implementation of the structural properties. Like the theorem prover in Chapter 4, this is a nice example of using a dependently typed programming language as its own

```
reifyn : ∀⇛ neu ⊃ (neuC # ⊃ expC #) ⊃ exp
reifyn Ψ (▷ x) σ = ▷ (σ x)
reifyn Ψ (napp · (n, s)) σ = app · (reifyn Ψ n σ, reify Ψ s σ)

reify : ∀⇛ sem ⊃ (neuC # ⊃ expC #) ⊃ exp
reify Ψ (slam · φ) σ =
  lam · (strengthen/anywhere exp (iS i0)
           (reify (Ψ, , neuC, , expC)
                  (weaken/irrel sem _ (φ [neuC] _ (neut · (▷ i0))))
                  σ')) where
    σ' : < Ψ, , neuC, , expC > (neuC # ⊃ expC #)
    σ' (iS i0) = i0
    σ' (iS (iS x)) = iS (iS (σ x))
reify Ψ (neut · n) σ = reifyn Ψ n σ
reify Ψ (▷ x) σ = impossible x
```

```
eval : ∀⇛ exp ⊃ ((expC #) ⊃ sem) ⊃ sem
eval Ψ (▷ x) σ = σ x
eval Ψ (app · (e1, e2)) σ = appsem _ (eval Ψ e1 σ) (eval Ψ e2 σ)
eval Ψ (lam · e) σ = slam · φ where
  φ : < Ψ > (∀⩽ neuC (sem ⊃ sem))
  φ Ψ' bnd s' = strengthen/middle sem Ψ'
                  (eval ((Ψ, , expC) + Ψ')
                        (weaken*/bounded exp Ψ' {neuC} bnd _ e)
                        σ') where
    σ' : < (Ψ, , expC) + Ψ' > (expC # ⊃ sem)
    σ' x with ListM.ln.splitappend Ψ' (Ψ, , expC) x
    ... | Inl x' = Sums.abort (ListM.ln.in-all _ bnd x')
    ... | Inr i0 = weaken/irrel/middle sem Ψ' s'
    ... | Inr (iS x') = weaken*/bounded sem Ψ' {neuC} bnd _
                          (weaken/irrel sem _ (σ x'))
```

Figure 5.4: Normalization by evaluation with a single context

tactic language.

The structural properties are implemented by instantiating a generic traversal for $< \Psi > \mathsf{A}$. The generic traversal has the following type:

$$\mathsf{map} : (\mathsf{A} : \mathsf{Type})\,\{\Psi\ \Psi' : \mathsf{Vars}\} \to (\mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi') \to < \Psi > \mathsf{A} \to < \Psi' > \mathsf{A}$$

This should be read as follows: for all A, Ψ, and Ψ', under the condition Co A Ψ Ψ', there is a map from terms of type A in Ψ to terms of type A in Ψ'.

$\mathsf{Co} : \mathsf{Type} \to \mathsf{Vars} \to \mathsf{Vars} \to \mathsf{Set}$ is a *variable relation*, a type-indexed family of relations between two contexts. Co is in fact a (module-level) parameter to the generic map; it must provide (1) a variable or term in Ψ' for each variable in Ψ that the traversal runs into; and (2) enough information to keep the traversal going inductively. We will instantiate Co with a specific relation for each traversal; e.g., for weakening with a variable of type D, Co will relate Ψ to $(\Psi, , \mathsf{D})$ under appropriate conditions on D and A.

For expository purposes, we present a slightly simplified version of the traversal first; the generalization is described with weakening below.

5.3.1 Compatibility

We ensure that Co provides the two pieces of information mentioned above using the notion of *compatibility*. Suppose that Co and Contra are variable relations. We say that Co and Contra are compatible iff there is a term

$$\mathsf{compat} : \{\mathsf{A} : \mathsf{Type}\}\,\{\Psi\ \Psi' : \mathsf{Vars}\} \to \mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi' \to \mathsf{Compat}\ \mathsf{A}\ \Psi\ \Psi'$$

where Compat is defined as follows:

$$\begin{aligned}
&\mathsf{Compat} : \mathsf{Type} \to \mathsf{Vars} \to \mathsf{Vars} \to \mathsf{Set}\\
&\mathsf{Compat}\ (\mathsf{D}\ \#)\ \Psi\ \Psi' = (\mathsf{D} \in \Psi) \to (\mathsf{D} \in \Psi')\\
&\mathsf{Compat}\ (\mathsf{D}^{+}\ \mathsf{D})\ \Psi\ \Psi' = (\{\mathsf{A} : \mathsf{Type}\} \to (\mathsf{c} : \mathsf{In}\Sigma\ (\mathsf{D} \Leftarrow \mathsf{A})) \to \mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi')\\
&\qquad\qquad \times (\{\mathsf{ch} : _\} \to (\triangleright\ \mathsf{D}\ \{\mathsf{ch}\}) \in \Psi \to < \Psi' > \mathsf{D}^{+}\ \mathsf{D})\\
&\mathsf{Compat}\ (\mathsf{A} \supset \mathsf{B})\ \Psi\ \Psi' = \mathsf{Contra}\ \mathsf{A}\ \Psi'\ \Psi \times \mathsf{Co}\ \mathsf{B}\ \Psi\ \Psi'\\
&\mathsf{Compat}\ (\Psi 0 \Rightarrow\!\!* \mathsf{A})\ \Psi\ \Psi' = \mathsf{Co}\ \mathsf{A}\ (\Psi + \Psi 0)\ (\Psi' + \Psi 0)\\
&\mathsf{Compat}\ (\mathsf{list}\ \mathsf{A})\ \Psi\ \Psi' = \mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi'\\
&\mathsf{Compat}\ (\Box\ \mathsf{A})\ \Psi\ \Psi' = \mathsf{Unit}\\
&\mathsf{Compat}\ (`1)\ \Psi\ \Psi' = \mathsf{Unit}\\
&\mathsf{Compat}\ (`0)\ \Psi\ \Psi' = \mathsf{Unit}\\
&\mathsf{Compat}\ (\mathsf{A} \otimes \mathsf{B})\ \Psi\ \Psi' = \mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi' \times \mathsf{Co}\ \mathsf{B}\ \Psi\ \Psi'\\
&\mathsf{Compat}\ (\mathsf{A} \oplus \mathsf{B})\ \Psi\ \Psi' = \mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi' \times \mathsf{Co}\ \mathsf{B}\ \Psi\ \Psi'\\
&\mathsf{Compat}\ (\forall \mathsf{c}\ \tau)\ \Psi\ \Psi' = (\Psi 0 : _) \to \mathsf{Co}\ (\tau\ \Psi 0)\ \Psi\ \Psi'\\
&\mathsf{Compat}\ (\forall \leqslant \mathsf{D}\ \mathsf{A})\ \Psi\ \Psi' = (\Psi 0 : _) \to \mathsf{AllEq}\ \Psi 0\ \mathsf{D} \to \mathsf{Co}\ \mathsf{A}\ (\Psi + \Psi 0)\ (\Psi' + \Psi 0)\\
&\mathsf{Compat}\ (\exists \mathsf{c}\ \tau)\ \Psi\ \Psi' = (\Psi 0 : _) \to \mathsf{Co}\ (\tau\ \Psi 0)\ \Psi\ \Psi'\\
&\mathsf{Compat}\ (\exists \leqslant \mathsf{D}\ \mathsf{A})\ \Psi\ \Psi' = (\Psi 0 : _) \to \mathsf{AllEq}\ \Psi 0\ \mathsf{D} \to \mathsf{Co}\ \mathsf{A}\ (\Psi + \Psi 0)\ (\Psi' + \Psi 0)\\
&\mathsf{Compat}\ (\forall \Rightarrow \mathsf{A})\ \Psi\ \Psi' = (\Psi 0 : _) \to \mathsf{Co}\ \mathsf{A}\ (\Psi + \Psi 0)\ (\Psi' + \Psi 0)\\
&\mathsf{Compat}\ (\exists \Rightarrow \mathsf{A})\ \Psi\ \Psi' = (\Psi 0 : _) \to \mathsf{Co}\ \mathsf{A}\ (\Psi + \Psi 0)\ (\Psi' + \Psi 0)\\
&\mathsf{Compat}\ ([\Psi]\ *\ \mathsf{A})\ _\ _ = \mathsf{Unit}
\end{aligned}$$

Compat imposes certain conditions on Co and Contra. For example, for variable types $\mathsf{D}\ \#$, it says that $\mathsf{Co}\ (\mathsf{D}\ \#)\ \Psi\ \Psi'$ induces a map from variables of type D in Ψ to variables in Ψ'. For defined atoms $\mathsf{D}^+\ \mathsf{D}$, Compat says that $\mathsf{Co}\ (\mathsf{D}^+\ \mathsf{D})\ \Psi\ \Psi'$ induces a map from variables in Ψ to terms in Ψ', and that $\mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi'$ holds for every premise A of every constant inhabiting D. In all other cases, Compat provides enough information to keep the induction going in map below. This amounts to insisting that Co (or Contra) holds on the subexpressions of a type in all appropriate contexts. For example, the condition for $\Psi 0 \Rightarrow^* \mathsf{A}$ is that Co holds for A in the contexts extended with $\Psi 0$. As mentioned briefly in Section 5.1, $\forall\Rightarrow$ and $\exists\Rightarrow$ and $[\backslash\_]\ ^*$ are actually built-in types, so they can be handled specially by the structural property tactics, though their semantics is the same as if they were derived forms.

In the usual monadic traversals of syntax (Altenkirch and Reus, 1999), $\mathsf{Co}\ \backslash\_\ \Psi\ \Psi'$ is taken to be $(\mathsf{D} : \mathsf{VarType}) \to \mathsf{D} \in \Psi \to < \Psi' > \mathsf{D}$—i.e. a realization of every variable in Ψ as a term in Ψ'. In our setting, this does not suffice to define a traversal, because (1) it does not provide for the contravariant flip necessary to process the domains of admissibility functions and (2) it does not allow us to express a *conditional* traversal, where conditions on the types ensure that the traversal will only find certain variables, and thus that only those variables need realizations. Compatibility ensures that Co provides enough information for Contra to process the contravariant positions to the left of a computational arrow. Additionally, it permits conditional traversals: below, we will instantiate Co so that it is uninhabited for certain A.

5.3.2 Map

Suppose that Co and Contra are compatible, and assume a function

$$\mathsf{map'} : (\mathsf{A} : \mathsf{Type})\ \{\Psi\ \Psi' : \mathsf{Vars}\} \to (\mathsf{Contra}\ \mathsf{A}\ \Psi\ \Psi') \to < \Psi > \mathsf{A} \to < \Psi' > \mathsf{A}$$

that is the equivalent of map for the Contravariant positions.

Then we implement map in Figure 5.5. In the first and second cases, the compatibility of Co induces the map on variables that we need. In the third case, we pre-compose the function with map' and post-compose with map. In all other cases, map simply commutes with constructors, or stops early if it hits a boxed term.

5.3.3 Exchange/Contraction

Exchange and contraction are implemented by one instantiation of map. In this case, we take

$$\mathsf{Co}\ \mathsf{A}\ \Psi\ \Psi' = \mathsf{Contra}\ \mathsf{A}\ \Psi\ \Psi' = (\Psi \subseteq \Psi' \times \Psi' \subseteq \Psi)$$

where $\subseteq$ means every variable in one context is in the other. It is simple to show that these relations are compatible, because Co (a) provides the required action on variables directly and (b) ignores its type argument, so the compatibility cases for the type constructors are easy. Exchange is defined by instantiating the generic map with Co, where map' is taken be map itself, which works because $\mathsf{Co} = \mathsf{Contra}$.

```
map : (A : Type) {Ψ Ψ' : Vars} → (Co A Ψ Ψ') → < Ψ > A → < Ψ' > A
map (D⁺ Dat) co (▷ x) = (snd (compat co) x)
map (Dat #) co x = ((compat co) x)
map (A ⊃ B) co e = λ y → (map B (snd (compat co)) (e (map' A (fst (compat co)) y)))
map (Ψ0 ⇒* A) co e = map A (compat co) e
map (list A) co [] = []
map (list A) co (x :: xs) = map A (compat co) x :: map (list A) co xs
map (D⁺ Dat) co (_·_ {A} c e) = c · map A (fst (compat co) c) e
map (□ A) co e = e
map (‘1) co t = <>
map (‘0) co ()
map (A ⊗ B) {Ψ} {Ψ'} co (e1, e2) = (map A (fst (compat co)) e1,
                                    map B (snd (compat co)) e2)
map (A ⊕ B) co (Inl e1) = Inl (map A (fst (compat co)) e1)
map (A ⊕ B) co (Inr e1) = Inr (map B (snd (compat co)) e1)
map (∀c τ) co e = λ Ψ0 → map (τ Ψ0) ((compat co) Ψ0) (e Ψ0)
map (∀⩽ D A) co e = λ Ψ0 ev → map A ((compat co) Ψ0 ev) (e Ψ0 ev)
map (∃⩽ D A) co (Ψ0, ev, e) = (Ψ0, ev, map A ((compat co) Ψ0 ev) e)
map (∃c τ) co (Ψ0, e) = Ψ0, map (τ Ψ0) ((compat co) Ψ0) e
map (∀⇒ A) co e = λ Ψ0 → map A ((compat co) Ψ0) (e Ψ0)
map (∃⇒ A) co (Ψ0, e) = Ψ0, map A ((compat co) Ψ0) e
map ([Ψ] * A) co e = e
```

Figure 5.5: Map

5.3.4 Strengthening

Next, we define a traversal that strengthens away variables that, based on type information, cannot possibly occur. The invariant for strengthening is the following:[3]

```
Co : Type → Vars → Vars → Set
Co A Ψ Ψ' = Σ (λ (D : VarType) → Σ (λ (i : D ∈ Ψ)
                  → Check (irrel (un▷ D) A) × Id Ψ' (Ψ - i)))
```

Here i, a pointer into the initial context Ψ is the variable to be strengthened away; the propositional equality constraint represented by the Identity says that the final context Ψ' is the initial context with i removed. The type Check (irrel (un▷ D) A) computes to Unit when strengthening is possible, and Void when it is not. Here un▷ simply peels off the injection of a defined atom into a VarType.

The crucial property of irrel is that Check (irrel (un▷ D) (D^+ D)) computes to Void. This forbids strengthening a variable of type D out of a term of type D. This is necessary because we cannot satisfy the usual compatibility condition for (D^+ D), which would require mapping all variables—including the variable-to-be-strengthened i—to a term of type D that does not mention i.

More generally, Check (irrel (un▷ D) A) means that variables of type D can never be used to construct terms of type A, which ensures that strengthening never runs into variables of the type being strengthened. The function irrel D A is defined by traversing the graph structure of types (i.e., it unrolls the definitions of defined atoms) and checks ¬ (DefinedAtoms.eq D Dat) for each defined atom Dat it finds.

To account for contravariance, we must define strengthening simultaneously with weakening by irrelevant assumptions, which is similar. About 250 lines of Agda code shows that these two relations together are compatible. Their traversals are then defined by instantiating map twice, mutually recursively—each is passed to the other as map' for the contravariant recursive calls.

5.3.5 Weakening

In addition to weakening by irrelevant types (e.g. weakening a nat with an exp), we can weaken by types that do not appear to the left of a computational arrow (e.g., weakening an exp with an exp).

For a simple version of weakening, the variable relation is similar to strengthening, but uses a different computed condition, and flips the role of Ψ and Ψ' (now Ψ' is bigger):

```
Co : Type → Vars → Vars → Set
Co A Ψ Ψ' = Σ λ (D : VarType) →
  Σ λ (i : D ∈ Ψ') →
    Check (canWeaken (un▷ D) A) × Id Ψ (Ψ' - i)
```

The function canWeaken is a different graph traversal than before: we check irrel (un▷ D) A for the left-hand side of each computational arrow A ⊃ B. Weakening can then be defined

[3]For concision, we suppress some details arising from the implementation of irrel, which takes a visited list as an extra argument; see the companion code for details.

using strengthening in contravariant positions, as irrel is exactly the condition that strengthening requires.

This suffices for a simple version of weakening. However, we can be more clever, and observe that types of the form $\forall\Rightarrow$ A are *always* weakenable, because their proofs are explicitly parametrized over arbitrary extensions of the context. Similarly, $\forall\leqslant$ C A is weakenable with any context composed entirely of C's. Capitalizing on this observation requires a slight generalization of the traversal described above: computationally, weakening $\forall\Rightarrow$ A does not recursively traverse the proof of A, like map usually does, but stops the traversal and instantiates the context quantifier appropriately. Thus, our actual implementation of map is parametrized so that, for each type A, either it is given sufficient information to transform A directly (a function $< \Psi > \mathsf{A} \rightarrow < \Psi' > \mathsf{A}$), or it has enough information to continue recursively, as in the compatibility conditions described above. We use the former only for weakening the quantifiers (map $< \Psi$ - i $>$ ($\forall\Rightarrow$ A) to $< \Psi >$ ($\forall\Rightarrow$ A)). We refer the reader to our Agda code for details. All told, weakening takes about 210 lines of Agda code to define and prove compatible.

5.3.6 Substitution

Substitution is similar to weakening and strengthening. Its invariant has the same form, using a condition canSubst (un$\triangleright$ D) A. This condition ensures two things: (1) that D is irrelevant to the left-hand-sides of any computational arrow, so that substitution can be defined using weakening-with-irrelevant-assumptions in the contravariant position, and (2) that D is weakenable with all variable types bound by A, so that the term being plugged in for the variable can be weakened as substitution goes under binders. Substitution takes about 220 lines to define and prove compatible.

5.4 Discussion

The main point of this chapter is to show that we can embed in Agda a logical framework that allows mixing of admissibility and derivability. These mixed definitions are not possible in LF-based systems, or in other tools or frameworks for representing binding in MLTT (Ambler et al., 2002; Capretta and Felty, 2007; Chlipala, 2007; Hickey et al., 2006; Momigliano et al., 2007). Though they can be coded by hand in MLTT, our framework improves on this by providing datatype-generic implementations of the structural properties of the hypothetical judgement. Derivability is represented in a pronominal, contextual manner, so the type system can be used to reason about the scoping of variables. We have used the framework to program a number of examples, including a scope-correct version of the normalization-by-evaluation problem discussed by Shinwell et al. (2003). Mixed definitions are not as difficult to handle in systems based on nominal logic, where all functions assume an infinite set of names, and therefore anticipate extensions of the context—the failure of weakening does not occur. However, because contexts are not tracked in the type system, reasoning about scoping invariants in these systems requires an external specification logic (Pottier, 2007).

It is interesting to consider where our implementation using dependent de Bruijn indices stands relative to LF-based systems for programming with binding. We hope to have demon-

strated that, when equipped generically with the structural properties, dependent de Bruijn indices are *not that bad*. The reason is that recursive functions over syntax typically do one of three things when they hit a binder: make a recursive call in an extended context, apply a structural property such as substitution, or continue pattern-matching under the binder. Our framework supports all of these operations. That said, there are certainly some benefits of LF-based systems that we were unable to mimic directly. First, these systems' termination checkers are aware that the structural properties do not change the size of a term, but our Agda implementation does not make this obvious. Second, it is much easier to write and read pronominal variables with a named syntax. Third, LF-based systems have a more convenient syntax for applying the structural properties. For example, the syntax of weakening and strengthening is relatively heavy in our setting. In Twelf and Delphin, weakening is silent, and strengthening (including strengthen? used in the η-contraction example) is marked by saying which variables do occur, using a non-linear higher-order pattern. In our Agda implementation, weakening must be marked explicitly, and strengthening requires one to enumerate those variables that do not occur instead. In Beluga, weakening and strengthening are written as substitutions.

Achieving the same convenience is an interesting and important direction for future work. One option would be to try to do this inside of the type theory, by switching to a different implementation of derivability. For example, by using a named form for free variables, weakening would need to adjust the proof that all the free variables of a term are in a context, but would not change the term itself—and hopefully these proof obligations could be handled automatically. Of course, named free variables have their own problems, as judgements cannot be freely α-converted. An alternative would be to extend the proof assistant with special syntax for writing down de Bruijn terms—e.g. elaborating a Beluga-like syntax to de Bruijn form. A challenge is that, because Beluga distinguishes the LF level from the Beluga level, there is a natural place to put the substitutions that mediate between different scopes. Without this distinction, it would require some thought to decide what the syntax should be.

Of course, one way in which all of the LF-based systems outpace ours is that they support dependent types–generic judgements as well as hypotheticals. In the dependently typed case, even getting as far as we have gotten here is quite tricky, as we discuss in Part III.

Chapter 6

Logical Foundations

The research described in this chapter was conducted jointly with Noam Zeilberger and Robert Harper, and published in LICS 2008 (Licata et al., 2008).

In this chapter, we describe the origins of the logical framework described above. This provides a more abstract presentation, which is divorced from the details of the Agda embedding. Additionally, the formal system we use, intuitionistic higher-order focusing, is interesting in its own right, independently of the application. Moreover, the Agda formalization of this system is itself a nice example of mixing admissibility and derivability.

As we discussed with BL_0 above, polarity (Girard, 1993) is a way of dividing up the logical connectives into two camps, called *positive* and *negative*. A positive connective, such as disjunction, requires making choices on the right (inl or inr?) but is *invertible* on the left—the left rule can be applied eagerly, independently of the rest of the sequent. Dually, a negative connective, such as implication, requires choices on the left (what do I apply it to?), but is invertible on the right. In general, inductive types are positive, whereas coinductive types are negative. Focusing (Andreoli, 1992a) is a proof discipline that involves chaining together choice and inversion steps. It is unsurprising that all inversion steps can be applied eagerly. What is more surprising is that that choice steps of like-polarity connectives can also be chained together, without loss of completeness. A focused sequent calculus forces all inversions to be applied eagerly, and all choices for a string of like-polarity connectives to be made at once.

Operationally, polarity and focusing can be given an intuitive explanation in terms of pattern-matching (Zeilberger, 2008a): Values of positive polarity are introduced by choosing a constructor, and eliminated by pattern-matching against their constructors. On the other hand, values of negative polarity are introduced by pattern-matching against their destructors, and eliminated by choosing a destructor. This is why, for example, in ordinary functional programming, functions can be defined by pattern-matching, but it is impossible to pattern-match against a function (except with a variable pattern)—implication (meaning admissibility) is a negative connective. The resulting calculus resembles A-normal form, and makes evaluation order explicit in the program text. Restricting the calculus in this way is useful for proof search (it limits the non-determinism) and for proof equality (focused proofs are canonical representatives of certain equivalence classes (Zeilberger, 2009)). Additionally, describing a logic within the strictures of focusing generally improves one's understanding of it.

This raises the following question: what is the polarity of derivability? On the one hand, it has a positive character, as one can pattern-match against a derivability. On the other, it has a negative character, as a derivability can be eliminated by substitution (choosing a term to substitute). Our investigation of this question leads to the notion of pre-derivability described above: we take the positive aspect of derivability as fundamental, and define a connective $\Rightarrow$ that is eliminated by pattern-matching. The negative aspect is then a definable notion, which corresponds to implementing the structural properties. Moreover, because the definition of $\Rightarrow$ involves only the identity principle (use of an assumption) and pattern-matching, the calculus is compatible with situations where the structural properties do not hold.

While this story—pre-derivability is positive, admissibility is negative—is appealingly dichotomous, the reality is more complicated. Pre-derivability is not inherently positive, but, rather, maintains the polarity of whatever connective it is applied to. For example, when used in an inductive definition, pre-derivability stays positive—in contrast to admissibility, which forces a polarity shift. But when used around a negative connective, pre-derivability also stays negative—as we saw above, $D \Rightarrow (A \supset B)$ can be applied to an A in an extended context to produce a B in an extended context. We express this in a focused calculus by defining two connectives, positive $\Rightarrow$ and negative $\curlywedge$, which are the positive and negative versions of pre-derivability. These satisfy a "some/any" theorem (Pitts, 2003) which says that they are interprovable in an appropriate sense.

In our Agda implementation, we implemented the structural properties using generic programming in the meta-language. An ordinary presentation of a sequent-calculus does not provide this opportunity for integrating meta-programs into the language. However, Zeilberger's higher-order presentation of focusing does (Zeilberger, 2008a,b, 2009). The idea of higher-order focusing is to represent inversion phases, or pattern-matching, by meta-level admissibility functions from patterns to expressions. In particular, these meta-level admissibility functions may exploit meta-level implementations of the structural properties, which proceed by induction on types. Higher-order focusing is also an extremely clean and concise system, where connectives are specified by their patterns, from which both the focusing and inversion phases are derived. Moreover, it enables connective-independent proofs of cut and identity. In this work, we extend higher-order focusing from classical logic (Zeilberger, 2008a) and positive-only intuitionistic logic (Zeilberger, 2008b) to full intuitionistic logic. This has since been shown to be a subsystem of a logic that admits both classical and intuitionistic connectives (Zeilberger, 2010), but the direct presentation of the intuitionistic part is somewhat more accessible.

In Section 6.1, we present a focused sequent calculus for intuitionistic higher-order focusing, and show how it can be used to define pre-derivability. We define the identity and cut-elimination procedures, and prove they are total under assumptions about the class of connectives. In Section 6.2, we show an Agda representation of this sequent calculus.

6.1 Sequent Calculus

A higher-order focused sequent calculus is defined in two stages. First, the polarized connectives are defined by axiomatizing the structure of *patterns.* Positive connectives are defined by *constructor patterns,* and negative connectives by *destructor patterns.* Second, there is a general focusing framework that is independent of the particular connectives of the logic.

We begin by defining a focused sequent calculus for polarized intuitionistic logic, including the simple structure of patterns and the general focusing rules.

6.1.1 Simple contexts and patterns

We write X^+, Y^+, Z^+ and X^-, Y^-, Z^- to stand for positive and negative propositional variables (atomic propositions), and A^+, B^+, C^+ and A^-, B^-, C^- to stand for arbitrary positive and negative formulas. We use α to range over *assumptions* X^+ or C^-, and dually γ to range over *conclusions* X^- or C^+. A *linear context* Δ is a list of assumptions.

The positive connectives are defined through the judgement $\Delta \Vdash C^+$, which corresponds to applying only linear right-rules to show C^+ from Δ. For example, the rules for atoms, conjunction, and disjunction are as follows:

$$\frac{}{X^+ \Vdash X^+} \quad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash B^+}{\Delta_1, \Delta_2 \Vdash A^+ \otimes B^+} \quad \frac{\Delta \Vdash A^+}{\Delta \Vdash A^+ \oplus B^+} \quad \frac{\Delta \Vdash B^+}{\Delta \Vdash A^+ \oplus B^+}$$

Foreshadowing the Curry-Howard interpretation, we refer to derivations of this judgement as *constructor patterns;* linearity captures the restriction familiar from functional programming that a pattern binds a variable just once.

Negative connectives are defined by $\Delta \Vdash C^- > \gamma$, which corresponds to using linear left-rules to decompose C^- into the conclusion γ. A proof term for this judgement is a *destructor pattern*, which gives the shape of an elimination context (continuation) for negative types:

$$\frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash B^- > \gamma}{\Delta_1, \Delta_2 \Vdash A^+ \to B^- > \gamma} \quad \frac{\Delta \Vdash A^- > \gamma}{\Delta \Vdash A^- \& B^- > \gamma} \quad \frac{\Delta \Vdash B^- > \gamma}{\Delta \Vdash A^- \& B^- > \gamma}$$

Observe that a destructor pattern for $A^+ \to B^-$ includes a constructor pattern for A^+, as well as a destructor pattern for B^-, matching the possible observations on a function type. We have adopted linear logic notation by writing $\otimes$ for positive and $\&$ for negative conjunction. In the present setting, both of these connectives encode ordinary intuitionistic conjunction with respect to provability, but they have different proof terms: positive conjunction is introduced by an eager pair whose components are values, and eliminated by pattern-matching against both components; negative conjunction is eliminated by projecting one of the components, and introduced by pattern-matching against either possible observation, i.e. by a lazy pair.

6.1.2 Focusing Judgements

In Figure 6.1, we present the focusing rules. In these rules, Γ stands for a sequence of linear contexts Δ, but Γ itself is treated in an unrestricted manner (i.e., variables are bound once in a pattern, but may be used any number of times within the pattern's scope).

The first two judgements concern the positive connectives. The judgement $\Gamma \vdash [C^+]$ defines right-focus on a positive formula, or *positive values*: a positive value is a constructor pattern under a substitution for its free variables. Focus judgements make choices: to prove C^+ in focus, it is necessary to choose a particular shape of value by giving a constructor pattern, and then satisfy the pattern's free variables. Values are eliminated with the left-inversion judgement

Assumption	α	::=	$X^+ \mid C^-$
Conclusion	γ	::=	$X^- \mid C^+$
Linear context	Δ	::=	$\cdot \mid \Delta, \alpha$
Unrestricted context	Γ	::=	$\cdot \mid \Gamma, \Delta$

Right Focus $\boxed{\Gamma \vdash [C^+]}$

$$\frac{\Delta \Vdash C^+ \quad \Gamma \vdash \Delta}{\Gamma \vdash [C^+]}$$

Left Inversion $\boxed{\Gamma \vdash \gamma_0 > \gamma}$

$$\frac{}{\Gamma \vdash X^- > X^-} \qquad \frac{(\Delta \Vdash C^+ \longrightarrow \Gamma, \Delta \vdash \gamma)}{\Gamma \vdash C^+ > \gamma}$$

Left Focus $\boxed{\Gamma \vdash [C^-] > \gamma}$

$$\frac{\Delta \Vdash C^- > \gamma_0 \quad \Gamma \vdash \Delta \quad \Gamma \vdash \gamma_0 > \gamma}{\Gamma \vdash [C^-] > \gamma}$$

Right Inversion $\boxed{\Gamma \vdash \alpha}$

$$\frac{(\Delta \Vdash C^- > \gamma \longrightarrow \Gamma, \Delta \vdash \gamma)}{\Gamma \vdash C^-} \qquad \frac{X^+ \in \Gamma}{\Gamma \vdash X^+}$$

Neutral $\boxed{\Gamma \vdash \gamma}$

$$\frac{\Gamma \vdash [C^+]}{\Gamma \vdash C^+} \qquad \frac{C^- \in \Gamma \quad \Gamma \vdash [C^-] > \gamma}{\Gamma \vdash \gamma}$$

Assumptions $\boxed{\Gamma \vdash \Delta}$

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta \quad \Gamma \vdash \alpha}{\Gamma \vdash \Delta, \alpha}$$

Figure 6.1: Focusing rules

$\Gamma \vdash \gamma_0 > \gamma$, which defines a *positive continuation* by case-analysis. Inversion steps respond to all possible choices that the corresponding focus step could make: the rule for C^+ quantifies over all constructor patterns for that formula, producing a result in each case. By convention, we tacitly universally quantify over metavariables such as Δ that appear first in a judgement that is universally quantified, so in full the premise reads "for all Δ, if $\Delta \Vdash C^+$ then $\Gamma, \Delta \vdash \gamma$." The positive connectives are thus introduced by choosing a value (focus) and eliminated by continuations that are prepared to handle any such value (inversion). For atoms, the only case-analysis is the identity.

The next two judgements concern the negative connectives, where the relationship between introduction/elimination and focus/inversion is reversed. A negative formula is *eliminated* by the left-focus judgement $\Gamma \vdash [C^-] > \gamma$, which chooses how to observe C^- by giving a *negative continuation*. A negative continuation consists of a destructor pattern, a substitution, and a case-analysis. The destructor pattern and substitution decompose a negative type C^- to some conclusion γ_0, for instance a positive type C^+. However, it may take further case-analysis of this positive type to reach the desired conclusion γ. Dually, negative types are *introduced* by inversion, which responds to left-focus by giving sufficient evidence to support all possible observations. The right-inversion judgement $\Gamma \vdash \alpha$, where assumptions α are negative formula or positive atoms, specifies the structure of a *negative value*. A negative value for C^- must show that for all destructors of C^-, the conclusion is justified by the variables bound by the patterns in it.

The judgement $\Gamma \vdash \gamma$, defines a neutral sequent, or an *expression*: from a neutral sequent, one can either right-focus and return a value, or left-focus on an assumption in Γ and apply a negative continuation to it. Finally, a *substitution* $\Gamma \vdash \Delta$ provides a negative value for each hypothesis.

Instantiating these rules using the above pattern rules, we see that they give the expected derived rules for the connectives; e.g.:

$$\frac{\Gamma \vdash X^- \quad \Gamma \vdash Y^- \quad \Gamma \vdash Z^-}{\Gamma \vdash (X^- \& Y^-) \& Z^-} \qquad \frac{\Gamma, X^+ \vdash Z^- \quad \Gamma, Y^+ \vdash Z^-}{\Gamma \vdash (X^+ \oplus Y^+) \to Z^-}$$

6.1.3 Patterns for Pre-derivability

In Section 6.1.1, we gave a fixed set of rules for constructing the patterns of some simple connectives. We now generalize this by making a context of *rules* a pattern of the pattern judgement. A rule R takes the form $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$, where $A_1^+, \ldots, A_n^+$ are positive formulas and P is a *defined atom*. Rules are collected in a *rule context* Ψ. The rule context is carried through the pattern judgements: $\Delta \Vdash A^+$ becomes $\Delta \Vdash \langle \Psi \rangle\, A^+$ and $\Delta \Vdash A^- > \gamma$ becomes $\Delta \Vdash \langle \Psi \rangle\, A^- > \gamma$. The rule context represents both the signature of constructors, and the local assumptions introduced by binders. This encompassed both $\mathsf{In}\Sigma$ and Ψ from Chapter 5, but is more general, because binders can locally introduce higher-order rules.

A rule $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+ \in \Psi$ can be applied to produce a constructor pattern for P:

$$\frac{\Delta_1 \Vdash \langle \Psi \rangle\, A_1^+ \quad \ldots \quad \Delta_n \Vdash \langle \Psi \rangle\, A_n^+}{\Delta_1, \ldots, \Delta_n \Vdash \langle \Psi \rangle\, P}$$

Note that rules are unrestricted, in the sense that they can be applied an arbitrary number of times while constructing a pattern. This pattern typing rule corresponds to the definition of the datatype `Data` in Chapter 5.

Pos. formula	A^+	::=	$X^+ \mid \downarrow A^- \mid 1 \mid A^+ \otimes B^+ \mid 0 \mid A^+ \oplus B^+ \mid P \mid R \Rightarrow B^+$
Rule	R	::=	$P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$
Neg. formula	A^-	::=	$X^- \mid \uparrow A^+ \mid A^+ \rightarrow B^- \mid \top \mid A^- \& B^- \mid R \curlywedge B^-$
Rule Context	Ψ	::=	$\cdot \mid \Psi, R$
Contextual form.	$C^\pm$	::=	$\langle \Psi \rangle\, A^\pm$

$$\boxed{\Delta \Vdash \langle \Psi \rangle\, A^+}$$

$$\frac{}{X^+ \Vdash \langle \Psi \rangle\, X^+} \qquad \frac{}{\langle \Psi \rangle\, A^- \Vdash \langle \Psi \rangle \downarrow A^-}$$

$$\frac{}{\cdot \Vdash \langle \Psi \rangle\, 1} \qquad \frac{\Delta_1 \Vdash \langle \Psi \rangle\, A^+ \quad \Delta_2 \Vdash \langle \Psi \rangle\, B^+}{\Delta_1, \Delta_2 \Vdash \langle \Psi \rangle\, A^+ \otimes B^+}$$

(no rule for 0)
$$\frac{\Delta \Vdash \langle \Psi \rangle\, A^+}{\Delta \Vdash \langle \Psi \rangle\, A^+ \oplus B^+} \qquad \frac{\Delta \Vdash \langle \Psi \rangle\, B^+}{\Delta \Vdash \langle \Psi \rangle\, A^+ \oplus B^+}$$

$$\frac{\Delta \Vdash \langle \Psi, R \rangle\, B^+}{\Delta \Vdash \langle \Psi \rangle\, R \Rightarrow B^+} \qquad \frac{P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+ \in \Psi \quad \Delta_1 \Vdash \langle \Psi \rangle\, A_1^+ \quad \ldots \quad \Delta_n \Vdash \langle \Psi \rangle\, A_n^+}{\Delta_1, \ldots, \Delta_n \Vdash \langle \Psi \rangle\, P}$$

$$\boxed{\Delta \Vdash \langle \Psi \rangle\, A^- > \gamma}$$

$$\frac{}{\cdot \Vdash \langle \Psi \rangle\, X^- > X^-} \qquad \frac{}{\cdot \Vdash \langle \Psi \rangle \uparrow A^+ > \langle \Psi \rangle\, A^+}$$

(no rule for $\top$)
$$\frac{\Delta \Vdash \langle \Psi \rangle\, A^- > \gamma}{\Delta \Vdash \langle \Psi \rangle\, A^- \& B^- > \gamma} \qquad \frac{\Delta \Vdash \langle \Psi \rangle\, B^- > \gamma}{\Delta \Vdash \langle \Psi \rangle\, A^- \& B^- > \gamma}$$

$$\frac{\Delta_1 \Vdash \langle \Psi \rangle\, A^+ \quad \Delta_2 \Vdash \langle \Psi \rangle\, B^- > \gamma}{\Delta_1, \Delta_2 \Vdash \langle \Psi \rangle\, A^+ \rightarrow B^- > \gamma} \qquad \frac{\Delta \Vdash \langle \Psi, R \rangle\, B^- > \gamma}{\Delta \Vdash \langle \Psi \rangle\, R \curlywedge B^- > \gamma}$$

Figure 6.2: Patterns

Pre-derivability is represented by two connectives, $\Rightarrow$ and $\curlywedge$:

$$\frac{\Delta \Vdash \langle \Psi, R\rangle\ B^+}{\Delta \Vdash \langle \Psi\rangle\ R \Rightarrow B^+} \qquad \frac{\Delta \Vdash \langle \Psi, R\rangle\ B^- > \gamma}{\Delta \Vdash \langle \Psi\rangle\ R \curlywedge B^- > \gamma}$$

Both connectives expand the rule context, introducing a scoped constructor of type R. The rule for $R \Rightarrow B^+$ builds a constructor pattern for B^+ under assumption of R and essentially (if we ignore structural punctuation) looks like an implication right-rule, while the rule for $R \curlywedge B^-$ builds a destructor pattern for B^- and looks like a conjunction left-rule. However, as we will see in Section 6.1.5, these connectives behave quite differently from ordinary implication and conjunction, in part due to their non-standard polarity.

Most of the remaining rules (see Figure 6.2) for the connectives of polarized logic are unremarkable, since they simply carry the rule context through unchanged. The "shift" connectives $\uparrow$ and $\downarrow$ deserve explanation, though. Following Girard (2001), these mark the boundary between positive and negative polarity, and correspondingly they mark the point where pattern-matching must end (Zeilberger, 2008a). Because the rule context can change during the course of pattern-matching, it is necessary to associate assumptions and conclusions with a specific rule context. We indicate this with *contextual formulas* $\langle \Psi\rangle\ A^+$ and $\langle \Psi\rangle\ A^-$, so that the rules for the shift connectives are:

$$\frac{}{\langle \Psi\rangle\ A^- \Vdash \langle \Psi\rangle \downarrow A^-} \qquad \frac{}{\cdot \Vdash \langle \Psi\rangle \uparrow A^+ > \langle \Psi\rangle\ A^+}$$

For example, pattern-matching on $\langle \Psi\rangle\ R \Rightarrow \downarrow A^-$ produces a variable of contextual type $\langle \Psi, \mathrm{R}\rangle\ A^-$. The same phenomenon occurs in Chapter 5, when pattern-matching on $\langle \Psi\rangle\ D \Rightarrow A$ yields an Agda variable of type $\langle \Psi, \mathrm{D}\rangle\ A$.

In spite of this richer notion of patterns, the generic focusing rules of Figure 6.1 remain unchanged if we adopt a notational sleight-of-hand: we now take C^+ and C^- to range over contextual formulas.

6.1.4 Identity and Cut

The context Ψ can be used to define arbitrary recursive types. For example, consider an atom D defined by one constant

$$\mathsf{d} : \mathsf{D} \Leftarrow \downarrow(\mathsf{D} \rightarrow \uparrow \mathsf{D})$$

D is essentially the recursive type $\mu X.X \rightarrow X$, which can be used to write non-terminating programs.

Because the rule context permits the definition of general recursive types, it should not be surprising that the identity and cut principles are not admissible in general. Through the Curry-Howard interpretation, however, we can still make sense of the identity and cut principles as corresponding, respectively, to the possibly infinite *processes* of η-expansion and β-reduction. We now state these principles, "prove" them by operationally sound but possibly non-terminating procedures, and then discuss criteria under which these proofs are well-founded.

PRINCIPLE 6.1.1: IDENTITY.

1. *(neg. identity) If* $C^- \in \Gamma$ *then* $\Gamma \vdash C^-$.
2. *(pos. identity)* $\Gamma \vdash C^+ > C^+$
3. *(identity substitution) If* $\Delta \subseteq \Gamma$ *then* $\Gamma \vdash \Delta$.

Procedure. The first identity principle reduces to the second and third as follows:

$$\dfrac{\forall(\Delta \Vdash C^- > \gamma) : \dfrac{C^- \in \Gamma \qquad \dfrac{\dfrac{}{\Delta \Vdash C^- > \gamma}\,\mathsf{ID3} \quad \dfrac{}{\Gamma, \Delta \vdash \Delta}\,\mathsf{ID3} \quad \dfrac{}{\Gamma \vdash \gamma > \gamma}\,\mathsf{ID2}}{\Gamma, \Delta \vdash [C^-] > \gamma}}{\Gamma, \Delta \vdash \gamma}}{\Gamma \vdash C^-}$$

The second identity reduces to the third as follows:

$$\dfrac{\forall(\Delta \Vdash C^+) : \dfrac{\dfrac{\dfrac{}{\Delta \Vdash C^+}\,\mathsf{ID3} \quad \dfrac{}{\Gamma, \Delta \vdash \Delta}\,\mathsf{ID3}}{\Gamma, \Delta \vdash [C^+]}}{\Gamma, \Delta \vdash C^+}}{\Gamma \vdash C^+ > C^+}$$

Finally, the third identity reduces to the first applied over all hypotheses $C^- \in \Delta$. □

PRINCIPLE 6.1.2: CUT.

1. *(neg. reduction) If* $\Gamma \vdash C^-$ *and* $\Gamma \vdash [C^-] > \gamma$ *then* $\Gamma \vdash \gamma$.
2. *(pos. reduction) If* $\Gamma \vdash [C^+]$ *and* $\Gamma \vdash C^+ > \gamma$ *then* $\Gamma \vdash \gamma$.
3. *(composition)*
   (a) *If* $\Gamma \vdash \gamma_0$ *and* $\Gamma \vdash \gamma_0 > \gamma$ *then* $\Gamma \vdash \gamma$.
   (b) *If* $\Gamma \vdash [C^-] > \gamma_0$ *and* $\Gamma \vdash \gamma_0 > \gamma$ *then* $\Gamma \vdash [C^-] > \gamma$.
   (c) *If* $\Gamma \vdash \gamma_1 > \gamma_0$ *and* $\Gamma \vdash \gamma_0 > \gamma$ *then* $\Gamma \vdash \gamma_1 > \gamma$.
4. *(substitution) For all six focusing judgements* J, *if* $\Gamma \vdash \Delta$ *and* $\Gamma, \Delta \vdash J$ *then* $\Gamma \vdash J$.

Procedure. Consider the first cut principle. The two derivations must take the following form:

$$\dfrac{\forall(\Delta \Vdash C^- > \gamma_0) : \quad \Gamma, \Delta \vdash \gamma_0}{\Gamma \vdash C^-} \qquad \dfrac{\Delta \Vdash C^- > \gamma_0 \quad \Gamma \vdash \Delta \quad \Gamma \vdash \gamma_0 > \gamma}{\Gamma \vdash [C^-] > \gamma}$$

By plugging $\Delta \Vdash C^- > \gamma_0$ from the right derivation into the higher-order premise of the left derivation, we obtain $\Gamma, \Delta \vdash \gamma_0$. Then $\Gamma \vdash \gamma_0$ by substitution with $\Gamma \vdash \Delta$, whence $\Gamma \vdash \gamma$ by composition with $\Gamma \vdash \gamma_0 > \gamma$. The case of positive reduction is analogous (but appeals only to substitution).

In all cases of composition, if $\gamma_0 = X^-$ then the statement is trivial. Otherwise, we examine the last rule of the left derivation. For the first composition principle, there are two cases: either the sequent was derived by right-focusing on the conclusion $\gamma_0 = C^+$, or else by left-focusing on some hypothesis $C^- \in \Gamma$. In the former case, we immediately appeal to positive reduction. In the latter case, we apply the second composition principle, which in turn reduces to the third, which then reduces back to the first.

Likewise, to show substitution we examine the rule concluding $\Gamma, \Delta \vdash J$. Dually to the composition principle, the only interesting case is when the sequent was derived by left-focusing on $C^- \in \Delta$, wherein we immediately apply a negative reduction.

□

Observe that we have made no mention of particular connectives or rule contexts, instead reasoning uniformly about focusing derivations. As we alluded to above, however, in general these procedures are not terminating. Here we state sufficient conditions for termination. They are stated in terms of a *strict subformula ordering,* a more abstract version of the usual structural subformula ordering.

DEFINITION 6.1.3: STRICT SUBFORMULA ORDERING. *We define an ordering $C_1^\pm \sqsupset C_2^\pm$ between contextual formulas as the least transitive relation closed under the following properties:*

- *If $\Delta \Vdash C_1^- > \gamma$ and $C_2^- \in \Delta$ then $C_1^- \sqsupset C_2^-$*
- *If $\Delta \Vdash C_1^- > \gamma$ and $C_2^+ = \gamma$ then $C_1^- \sqsupset C_2^+$*
- *If $\Delta \Vdash C_1^+$ and $C_2^- \in \Delta$ then $C_1^+ \sqsupset C_2^-$*

For any contextual formula $C^\pm$, we define $\sqsupset_C$ to be the restriction of $\sqsupset$ to formulas below $C^\pm$.

The strict subformula ordering does not mention atoms X^+ or X^-, since they only play a trivial role in identity and cut.

DEFINITION 6.1.4: WELL-FOUNDED FORMULAS. *We say that a contextual formula $C^\pm$ is well-founded if $\sqsupset_C$ is well-founded.*

PROPOSITION 6.1.5. *Positive and negative identity are admissible on well-founded formulas.*

Proof. By inspection of the above procedure. Positive and negative identity are proved by mutual induction using the order $\sqsupset_C$, with a side induction on the length of Δ to show substitution identity. □

PROPOSITION 6.1.6. *Positive and negative reduction are admissible on well-founded formulas.*

Proof. By inspection of the above procedure. Positive and negative reduction are proved by mutual induction using the order $\sqsupset_C$, with a side induction on the left derivation to show composition, and a side induction on the right derivation to show substitution. □

DEFINITION 6.1.7: PURELY POSITIVE RULES. *A rule R is called* purely positive *if it contains no shifted negative formulas $\downarrow A^-$ as premises (or structural subformulas of premises). For example,* $\mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp})$ *is pure, but* $\mathsf{D} \Leftarrow \Downarrow(\mathsf{D} \to \uparrow D)$ *is not.*

LEMMA 6.1.8. *Suppose $\langle\Psi\rangle\ A^\pm$ contains only purely positive rules (i.e., in Ψ, or as structural subformulas of $A^\pm$). Then $\langle\Psi\rangle\ A^\pm$ is well-founded.*

Proof. By induction on the structure of $A^\pm$. Every pattern typing rule (recall Figure 6.2) examines only structural subformulas of $A^\pm$, except when $A^+ = P$. But any P defined by purely positive rules $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$ in fact has *no* strict subformulas, since the Δ_i such that $\Delta_i \Vdash \langle\Psi\rangle\ A_i^+$ can contain only atomic formulas X^+. □

The restriction to pure rules precludes premises involving admissibility. However, it includes all inference rules definable in the LF logical framework, generalizing Schroeder-Heister's proof of cut-elimination for the fragment of definitional reflection with $\to$-free rules (Schroeder-Heister, 1993) (since purely positive rules do *not* exclude $\Rightarrow$'s). Moreover, as we explained, the identity and cut principles are always operationally meaningful, even in the presence of arbitrary recursive types. Technically, we could adopt a coinductive reading of the focusing rules (cf. (Girard, 2001)), in which case identity is always productive, and cut-elimination is a partial

operation that attempts to build a cut-free proof bottom-up. We conjecture that cut-elimination is total given a positivity restriction for rules.

6.1.5 Shock therapy

In §6.2 of "Locus Solum", Girard (2001) considers several "shocking equalities"—counterintuitive properties of the universal and existential quantifiers that emerge when they are given non-standard polarities. For example, positive $\forall$ commutes under $\oplus$, while negative $\exists$ commutes over $\&$. In our setting, $\Rightarrow$ behaves almost like a positive universal quantifier, and $\curlywedge$ almost like a negative existential. These would become real quantifiers in an extension to dependent types. And indeed, we can reproduce analogues of these commutations.

DEFINITION 6.1.9. *For two positive contextual formulas C_1^+ and C_2^+, we say that $C_1^+ \lesssim C_2^+$ if $\cdot \vdash C_1^+ > C_2^+$. For negative C_1^- and C_2^-, we say $C_1^- \lesssim C_2^-$ if $C_1^- \vdash C_2^-$. We write $C_1^\pm \approx C_2^\pm$ when both $C_1^\pm \lesssim C_2^\pm$ and $C_2^\pm \lesssim C_1^\pm$. These relations are extended to (non-contextual) polarized formulas if they hold under all rule contexts.*

PROPOSITION 6.1.10: "SHOCKING" EQUALITIES.

1. $R \Rightarrow (A^+ \oplus B^+) \approx (R \Rightarrow A^+) \oplus (R \Rightarrow B^+)$
2. $(R \curlywedge A^-)\&(R \curlywedge B^-) \approx R \curlywedge (A^-\&B^-)$

Proof. Immediate—indeed, in each case, both sides have an isomorphic set of patterns. □

Why are these equalities shocking? If we ignore polarity and treat all the connectives as ordinary implication, disjunction, and conjunction, then (2) is reasonable but (1) is only valid in classical logic. And if we interpret $\Rightarrow$ and $\curlywedge$ as $\forall$ and $\exists$, then both equations are shockingly anticlassical:

1. $\forall x.(A \oplus B) \approx (\forall x.A) \oplus (\forall x.B)$
2. $(\exists x.A)\&(\exists x.B) \approx \exists x.(A\&B)$

On the other hand, from a computational perspective, these equalities are quite familiar. For example, (1) says that a value of type $A \oplus B$ with a free variable is either the left injection of an A with a free variable or the right injection of a B with a free variable.

We can state another pair of surprising equivalences *between* the connectives $\Rightarrow$ and $\curlywedge$ under polarity shifts:

PROPOSITION 6.1.11: SOME/ANY.

1. $\downarrow(R \curlywedge A^-) \approx R \Rightarrow \downarrow A^-$
2. $\uparrow(R \Rightarrow A^+) \approx R \curlywedge \uparrow A^+$

Again, this coincidence under shifts is not *too* surprising, since it recalls the some/any quantifier $\text{И}x.A$ of nominal logic (Pitts, 2003), as well as the self-dual ∇ connective of Miller and Tiu (2003). $\text{И}x.A$ can be interpreted as asserting either that A holds for some fresh name, or for *all* fresh names—with both interpretations being equivalent.

6.1.6 Relationship to modal logic

Though we did not originally recognize it as such, the above system in fact corresponds to a hybrid modal logic presented in the style of Simpson (Simpson, 1993), where the rule context Ψ is the world. We can define the hybrid connectives $A@\Psi$ (which we called $[\Psi]^* A$ above) and $\downarrow \psi.A$ by pattern rules as follows:

$$\frac{\Delta \Vdash \langle \Psi' \rangle\ A^+}{\Delta \Vdash \langle \Psi \rangle\ A^+@^+\Psi'} \qquad \frac{\Delta \Vdash \langle \Psi \rangle\ [\Psi/\psi]A^+}{\Delta \Vdash \langle \Psi \rangle \downarrow^+\psi.A^+} \qquad \frac{\Delta \Vdash \langle \Psi' \rangle\ A^- > \gamma}{\Delta \Vdash \langle \Psi \rangle\ A^-@^-\Psi' > \gamma} \qquad \frac{\Delta \Vdash \langle \Psi \rangle\ [\Psi/\psi]A^- > \gamma}{\Delta \Vdash \langle \Psi \rangle \downarrow^-\psi.A^- > \gamma}$$

Then $R \Rightarrow A^+$ is definable as $\downarrow \psi.A^+@(\psi, R)$: the pattern rule for $\Rightarrow$ has the same premise as the single derived rule for this composition of connectives. A similar result holds for $\curlywedge$.

This connection with modal logic sheds more light on the some/any theorem described above, once we observe that both $@$ and $\downarrow$ also satisfy this equivalence under shifts. Our explanation for this ambipolar nature of $\curlywedge$, $@$, and $\downarrow$ is that they are "macro" connectives, which can in fact be interpreted away—as we exploited in our Agda implementation above. The interpretation of each of these connectives ($R \Rightarrow A$, $A@\Psi$, $\downarrow \psi.A$) is defined to be the interpretation of A in a modified context or under a substitution, so, after interpretation, the outer structure of the proposition is the structure of A.

This connection with modal logic also provides new insight into the Agda implementation of our framework: the interpretation of Types as functions from Ctx → Set corresponds to implementing the Kripke semantics of the modal logic. This observation suggests a way of representing modal logic in a dependently typed programming language, which we have exploited in a formalization of the ML5 programming language (Licata and Harper, 2010).

6.2 Agda Representation of Focusing

In this section, we show how to represent higher-order focusing in Agda. Higher-order focusing is itself a nice example of mixing admissibility and derivability: the context of negative assumptions Γ requires derivability, but the higher-order representation of inversion requires admissibility. For simplicity, we show only non-contextualized patterns, though the contextualized case is no more difficult to represent.

In Figure 6.3, we represent the types as a straightforward inductive definition. We combine the mutually inductive definitions of positive and negative types into one inductive definition indexed by a polarity; this makes it easier to define operations that work on both polarities of types.

Next, we represent contexts:

```
data Hyps : Set where
  ·      : Hyps
  _,_    : Hyps → Hyps → Hyps
  _true⁻ : Type⁻ → Hyps
  _atom⁺ : Atom → Hyps
```

```
Atom : Set
Atom = String

data Pol : Set where
  Pos : Pol
  Neg : Pol

mutual
  Type⁻ : Set
  Type⁻ = Type Neg

  Type⁺ : Set
  Type⁺ = Type Pos

  data Type : (p : Pol) → Set where
    X⁺    : Atom → Type⁺
    ↓     : Type⁻ → Type⁺
    ‘1    : Type⁺
    _⊗_   : Type⁺ → Type⁺ → Type⁺
    ‘0    : Type⁺
    _⊕_   : Type⁺ → Type⁺ → Type⁺
    nat   : Type⁺

    X⁻    : Atom → Type⁻
    ↑     : Type⁺ → Type⁻
    _⊃_   : Type⁺ → Type⁻ → Type⁻
    ⊤     : Type⁻
    _&_   : Type⁻ → Type⁻ → Type⁻
```

Figure 6.3: Agda Representation of Polarized Types

```
Ctx : Set
Ctx = List Hyps

data Conc : Set where
  _true⁺ : Type⁺ → Conc
  _atom⁻ : Atom → Conc

data _∈Δ_ : (Δ : Hyps) → Hyps → Set where
  iX : ∀ {X} → (X atom⁺) ∈Δ (X atom⁺)
  iT : ∀ {A⁻ : Type⁻} → (A⁻ true⁻) ∈Δ (A⁻ true⁻)
  i1 : ∀ {Δ Δ1 Δ2} → Δ ∈Δ Δ1 → Δ ∈Δ (Δ1, Δ2)
  i2 : ∀ {Δ Δ1 Δ2} → Δ ∈Δ Δ2 → Δ ∈Δ (Δ1, Δ2)

data _∈∈_ : Hyps → Ctx → Set where
  _,_ : ∀ {Δ Δ' Γ} → Δ' ∈ Γ → Δ ∈Δ Δ' → Δ ∈∈ Γ
```

Contexts are lists of hypotheses, where hypotheses are join-lists of atomic assumptions of the form A true$^+$ and X atom$^-$. Dually, conclusions are of the form A true$^+$ or X atom$^-$—intuitionistic logic admits only a single conclusion. The type $\in\Delta$ represents de Bruijn indices into Δ, and the type $\in\in$ represents a de Bruijn index $\Delta \in \Gamma$ paired with an index into Δ.

Patterns are defined in Figure 6.4 by a straightforward inductive definition, directly transcribing the above rules.

We show the focusing rules in Figure 6.5. We define a sum type Judge for the right-hand sides of all of the focusing judgements, so that we can define the focusing rules with a single judgement $\Gamma \vdash \mathsf{J}$. It is convenient to break out $\Gamma \vdash \gamma > \gamma'$ as a separate judgement, and to move the right rule for positive atoms from the inversion judgement (in the rules in Section 6.1) to the substitution judgement. This way the value judgements (right focus and inversion) and the continuation judgements (left focus and inversion) both operate on a *type*, rather than an assumption or conclusion. This also necessitates the new rule UseT, which passes from Use to inversion. Other than these changes, each datatype constructor is a straightforward transcription a sequent rule from Section 6.1. Val$^-$ and Cont$^+$ use both admissibility (to quantify over patterns) and derivability (to extend the context with the negative variables they produce).

In the companion code, we give a simple implementation of the identity and cut principles described above. We do not prove termination of these functions, though one could do so by proving well-foundedness of the structural subformula order.

6.3 Discussion

Zeilberger (Zeilberger, 2009) argues that polarity and focusing provide a logical account of both evaluation order and pattern-matching. It is the latter that facilitated our original investigation of a proof theory mixing pre-derivability and admissibility: higher-order focusing allows a concise but high-level description of inductive types, off-loading the hard work of describing induction principles on meta-level admissibility functions. Its account of evaluation order makes focusing a promising technique for studying programming languages with effects—for example, in the presence of certain effects, focused proofs are normal forms of contextual equivalence classes (Zeilberger, 2009). However, we have not exploited this aspect of focused proofs in

```
data _⊩⊢_ : Hyps → Type⁺ → Set where
  Cx⁺ : ∀ {X} → (X atom⁺) ⊩⊢ (X⁺ X)
  Cx⁻ : ∀ {A⁻} → (A⁻ true⁻) ⊩⊢ (↓ A⁻)
  C<> : · ⊩⊢ ‘1
  Cpair : ∀ {Δ₁ Δ₂ A⁺ B⁺}
    → Δ₁ ⊩⊢ A⁺ → Δ₂ ⊩⊢ B⁺
    → (Δ₁, Δ₂) ⊩⊢ (A⁺ ⊗ B⁺)
  Cinl : ∀ {Δ A⁺ B⁺}
      → Δ ⊩⊢ A⁺
      → Δ ⊩⊢ (A⁺ ⊕ B⁺)
  Cinr : ∀ {Δ A⁺ B⁺}
      → Δ ⊩⊢ B⁺
      → Δ ⊩⊢ (A⁺ ⊕ B⁺)
  Czero : · ⊩⊢ nat
  Csucc : ∀ {Δ}
      → Δ ⊩⊢ nat
      → Δ ⊩⊢ nat
data _⊩⊢_>_ : Hyps → Type⁻ → Conc → Set where
  De⁻ : ∀ {X} → · ⊩⊢ (X⁻ X) > (X atom⁻)
  De⁺ : ∀ {A⁺} → · ⊩⊢ (↑ A⁺) > (A⁺ true⁺)
  Dapp : ∀ {Δ1 Δ2 A⁺ B⁻} {γ : Conc}
      → Δ1 ⊩⊢ A⁺ → Δ2 ⊩⊢ B⁻ > γ
      → (Δ1, Δ2) ⊩⊢ (A⁺ ⊃ B⁻) > γ
  Dfst : ∀ {Δ A⁻ B⁻} {γ : Conc}
      → Δ ⊩⊢ A⁻ > γ
      → Δ ⊩⊢ (A⁻ & B⁻) > γ
  Dsnd : ∀ {Δ A⁻ B⁻} {γ : Conc}
      → Δ ⊩⊢ B⁻ > γ
      → Δ ⊩⊢ (A⁻ & B⁻) > γ
```

Figure 6.4: Agda Representation of Patterns

```
data FocJudg : Set where
  Val  : {p : Pol} → Type p → FocJudg
  Cont : {p : Pol} → Type p → Conc → FocJudg
  Use  : Conc → Conc → FocJudg
  Sub  : Hyps → FocJudg
  Exp  : Conc → FocJudg

data _⊢_ : Ctx → FocJudg → Set where
  Val⁺ : ∀ {Γ Δ} {C⁺ : Type⁺}
       → Δ ⊪ C⁺ → Γ ⊢ Sub Δ
       → Γ ⊢ Val C⁺
  Val⁻ : ∀ {Γ} {C⁻ : Type⁻}
       → (∀ {Δ γ} → Δ ⊪ C⁻ > γ → Δ :: Γ ⊢ Exp γ)
       → Γ ⊢ Val C⁻
  Cont⁺ : ∀ {Γ γ} {C⁺ : Type⁺}
         → (∀ {Δ} → Δ ⊪ C⁺ → Δ :: Γ ⊢ Exp γ)
         → Γ ⊢ Cont C⁺ γ
  Cont⁻ : ∀ {Δ₀ γ₀ γ Γ} {A⁻ : Type⁻}
        → Δ₀ ⊪ A⁻ > γ₀ → Γ ⊢ Sub Δ₀ → Γ ⊢ Use γ₀ γ
        → Γ ⊢ Cont A⁻ γ
  UseX : ∀ {Γ X}
       → Γ ⊢ Use (X atom⁻) (X atom⁻)
  UseT : ∀ {Γ γ} {A⁺ : Type⁺}
         → Γ ⊢ Cont A⁺ γ
         → Γ ⊢ Use (A⁺ true⁺) γ
  SubX : ∀ {Γ X}
       → (X atom⁺) ∈∈ Γ
       → Γ ⊢ Sub (X atom⁺)
  SubT : ∀ {Γ} {C⁻ : Type⁻}
       → Γ ⊢ Val C⁻
       → Γ ⊢ Sub (C⁻ true⁻)
  ·    : ∀ {Γ}
       → Γ ⊢ Sub ·
  _,_  : ∀ {Δ1 Δ2 Γ}
       → Γ ⊢ Sub Δ1 → Γ ⊢ Sub Δ2
       → Γ ⊢ Sub (Δ1 , Δ2)
  R : ∀ {Γ} {C⁺ : Type⁺}
    → Γ ⊢ Val C⁺
    → Γ ⊢ Exp (C⁺ true⁺)
  L : ∀ {Γ γ} {A⁻ : Type⁻}
    → (A⁻ true⁻) ∈∈ Γ → Γ ⊢ Cont A⁻ γ
    → Γ ⊢ Exp γ
```

Figure 6.5: Agda Representation of Focusing Rules

this dissertation, where we have concentrated on derivability and admissibility in pure type theory. Thus, our initial investigation into mixing derivability and admissibility was exploiting the "pattern-matching" but not "evaluation order" aspects of focusing.

Another way to achieve the same ends is to use inductive types inside of an existing type theory—where someone has similarly already done the hard work of describing induction principles. This led to the approach described above in Chapter 5, which was more immediately practical, as it allowed us to exploit an existing proof assistant more directly. Programming in the encoded focused proof system described above offers no real advantages until one considers effects.

Part III

Directed Dependent Type Theory

Chapter 7

2-Dimensional Directed Type Theory

The research described in this part was conducted jointly with Robert Harper.

7.1 Motivation and Background

To make the embedded logical framework described above useful for programming with logics, rather than just syntax, we need general, as well as hypothetical, judgements. As discussed in Chapter 2, the structural properties of the generic judgements are more subtle than those of the hypothetical. For example, recall the rule for substitution:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]}\ \mathsf{subst}$$

This rule says that the substitution into the derivation proves the substitution into the context and judgement. When implemented with de Bruijn indices, weakening, exchange, and contraction exhibit similar phenomena: stating them for derivations requires a corresponding action on judgements.

To achieve these substitution principles, it is necessary that each inference rule commutes with the structural properties, in the sense that the substitution into the conclusion of the rule is the conclusion of the substitution into the premises. This permits substitution to be implemented by reapplying the rule to the substitution instances of the premises. For example, consider elimination for the universal quantifier:

$$\frac{\Gamma, x : \mathsf{term} \vdash A\ \mathsf{prop} \quad \Gamma \vdash \forall x.A\ \mathsf{true} \quad \Gamma \vdash t : \mathsf{term}}{\Gamma \vdash A[t/x]\ \mathsf{true}}$$

Suppose $\Gamma = y : \mathsf{term}$, and that s is a closed term. Then substituting for y in the premises gives:

$$\begin{array}{l} x : \mathsf{term} \vdash A[s/y]\ \mathsf{prop} \\ \forall x.(A[s/y])\ \mathsf{true} \\ t[s/y] : \mathsf{term} \end{array}$$

and we are to show that $A[t/x][s/y]$ is true. To put the substituted premises together to yield the conclusion, we need to know that substitutions *compose* in the following sense:

$$A[t/x][s/y] = A[s/y][t[s/y]/x]$$

This raises the question of which terms commute with substitution. Clearly datatype constructors do, and, as we have just argued, substitutions themselves do as well. What about functions defined recursively? If the index is an admissibility function, what does it mean to commute with substitution at higher type?

In this chapter, we develop a framework for answering these questions. Our approach is to view the structural properties as an instance of a much more general idea, that of *directed* phenomena in dependent type theory. This has several benefits: First, it gives a clear account of what constraints must be placed on the indices of structural judgements. Moreover, this account is not tied to a specific embedded logical framework. This is useful because, even in this dissertation, we have used a variety of notions of entailment, including standard hypotheticals (Chapter 5), as well as modal logic (in Chapter 3) and affine logic (in Chapter 4). Second, our approach suggests that with a little more work we will able to automatically equip all of these logics with the structural properties, just by writing down the datatype definitions.

7.1.1 Structural properties as functoriality

To illustrate what we mean by directed phenomena, we recast the structural properties in the language of category theory, following Altenkirch and Reus (1999); Fiore et al. (1999); Hofmann (1999). We assume the reader is familiar with categories, functors, and natural transformations, but will define the other concepts we need.

The universe of contextual Types defined in Chapter 5 are interpreted as functions $\mathsf{Ctx} \to \mathsf{Set}$. Our first observation is that the structural properties endow these functions with the structure of a *functor* from a category of contexts to the category of sets. To cover the structural properties of weakening, exchange, and contraction, we can define Ctx to be the category where an object is a context Ψ, and a morphism $\longrightarrow \Psi_1\Psi_2$ is a variable-for-variable substitution, represented using the type $\Psi_1 \subseteq \Psi_2$ defined above. A functor $Ctx \longrightarrow Sets$ consists of a family of sets F_Ψ, which preserves context maps: if $\Psi \subseteq \Psi'$ then there is a function $F_\Psi \to F_{\Psi'}$. Of course, functoriality is exactly the definition of weakening (for a whole context at once), as used e.g. in Chapter 3.

The Types of our universe show that the collection of functors $Ctx \longrightarrow Sets$ is closed under various type-forming operations, whose action on morphisms is given in the definition of map.

- $F \otimes G$ is the product functor, whose action on an object Ψ is given by the product of F_Ψ and G_Ψ, and action on morphisms is given componentwise.
- $F \oplus G$ is the sum functor, whose action on an object Ψ is given by the sum of F_Ψ and G_Ψ, and action on morphisms is given by case-analysis.
- $\Box A$ is a constant functor, with trivial action on morphisms.
- $D\#$ says that the type $D \in -$ is functorial, with action on morphisms given by applying the substitution.
- D^+D says that the recursive type $\mathsf{Data}\ -\ D$ is functorial, with action on morphisms given for variables, by applying the substitution, or for constants, recursively.

In contrast, the type $F \supset G$ defines an action on objects (Ψ goes to $F_\Psi \to G_\Psi$), but is *not* functorial, because the $\to$ is *contravariant* in its domain. Given a contravariant functor $F : Ctx^{\mathsf{op}} \longrightarrow Sets$ and a covariant functor $G : Ctx \longrightarrow Sets$, we can form $F \supset G$ with action on morphisms given by pre- and post-composition. However, given a covariant functor for the domain, then the intended pre-composition faces the wrong direction (we have $F_{\Psi_1} \to F_{\Psi_2}$ when we need $F_{\Psi_2} \to F_{\Psi_1}$). This provides another explanation for why the structural properties fail for admissibility functions: the structural properties are implemented by the functorial actions of the type constructors, but $\supset$ does not determine a functor.

It also explains the co- and contravariant weave in the definition of the structural properties above: the tactics defined above sort out, after the fact, whether a type defines a functor, and out of what category. That is, the remaining structural properties can be defined by looking at different notions of morphism for the category of contexts. If we define $Ctx^{\geq D}$ to have, as morphisms $\Psi \longrightarrow \Psi'$, proofs that Ψ and Ψ' only differ by variables not subordinate to D, then a functor $Ctx^{\geq D} \longrightarrow Sets$ is a type that is weakenable and strengthenable by variables not subordinate to D. If we define the morphisms to be term-for-variable substitutions, then functoriality gives substitution as well as weakening, exchange, and contraction. Of course, these notions can be combined as well.

7.1.2 Structural Properties of the Generic Judgement

The advantage of this categorical view is that it extends cleanly to the dependently typed case. A simple type, like a type Prop of propositions, determines a functor $Ctx \to Sets$. An indexed judgement like $\Gamma \vdash A\ \mathsf{true}$ determines a functor $(\Sigma\ \Psi{:}Ctx.\ \mathsf{Prop}(\Psi)) \longrightarrow Sets$, whose action on objects gives the set of derivations, and action on morphisms corresponds exactly to the structural properties of the generic judgement. Thus, our next task is to understand Σ-types in category theory.

Given a category $\mathcal{C}$ and a functor $F : \mathcal{C} \longrightarrow Sets$, the *category of elements of F*, or *total category of the Grothendieck construction on F*, is the category-theoretic analogue of the type $\Sigma\ A{:}\mathcal{C}.\ F\ A$. It is written $\int_{\mathcal{C}} F$:

- an object of $\int_{\mathcal{C}} F$ is a pair (o, a) where $o \in \mathsf{Ob}\ \mathcal{C}$, and $a \in F(o)$.
- a morphism from (o, a) to (o', a') is a pair (c, f) where $c : o \longrightarrow_{\mathcal{C}} o'$ and f is a proof that $A(c)o = o'$ as elements of the set $A(o')$.
- $\mathsf{id}_{(o,a)} = (\mathsf{id}_o, \mathsf{refl})$. This is well-typed because $A(\mathsf{id}) = \mathsf{id}$, so the second component requires a proof that $a = a$.
- $(c, f) \circ (c', f') = (c \circ c', f \circ A(c)f')$. In the second component we write $\circ$ for transitivity on equality proofs; this composition proves the goal because A preserves compositions.

Thus, the set of objects of $\int_{\mathcal{C}} F$ is exactly $\Sigma\ A{:}\mathsf{Ob}\ C.\ \mathsf{Ob}\ (FA)$. A morphism is a morphism of the first component that is respected by the assignment of elements given by the second component. Specializing this definition to $\Sigma\ \Psi{:}Ctx.\ \mathsf{Prop}(\Psi)$, a morphism from (Ψ, A) to (Ψ', A') is a morphism $w : \Psi \longrightarrow \Psi'$ such that $A' = \mathsf{map}\, w\, A$. Thus, if a type family $\vdash$ is a functor $(\Sigma\ \Psi{:}Ctx.\ \mathsf{Prop}(\Psi)) \longrightarrow Sets$, then a context map $w : \Psi \longrightarrow \Psi'$ determines a function $\vdash(\Psi, A) \to \vdash(\Psi', \mathsf{map}\, w\, A)$. The substitution into the derivation proves the substitution into the

judgement, just as we desired.

This discussion motivates studying categories Ctx and $\int$ and functors from them into $Sets$. However, more generality is required. Consider a first-order logic with judgements of the form $\Omega; \Gamma \vdash A$, where Ω is an individual context, and Γ is dependent on Ω. As a first cut, we might represent this situation by a category $ICtx$ whose objects are Ω's and whose morphisms give their structural properties, as well as a functor $TCtx : ICtx \longrightarrow Sets$ giving the the truth contexts for each individual context. However, the truth contexts themselves should not be a set, but a category, with morphisms representing their structural properties. Thus, we need functors into Cat, the category of categories, not just $Sets$.

Second, when writing inference rules, we need to mention elements of these sets and categories (e.g. particular proposition constructors). These elements cannot be arbitrary objects, but, as discussed above, must commute with the structural properties. This condition falls out naturally when we consider a categorical interpretation not just of types, but also of terms.

However, once we are at the point where we are considering a categorical interpretation of both types and terms, what we are doing is of course *defining a programming language* with a category-theoretic semantics. This connects our work with recent investigations in *higher-dimensional type theory*.

7.1.3 Higher-dimensional type theory

One of the most cherished ideas in programming languages is the correspondence between logic, type theory, and category theory. Recent research has suggested a second holy trinity, between dependent type theory, higher-dimensional category theory, and homotopy theory (Awodey and Warren, 2009; Garner, 2009; Hofmann and Streicher, 1998; Lumsdaine, 2009; van den Berg and Garner, 2010; Voevodsky, 2010). The homotopy hypothesis (Baez and Shulman, 2007) postulates a correspondence between higher-dimensional categories and higher-dimensional homotopy types, and recent work has extended this to a correspondence with type theory.

On the type theoretic side, this correspondence suggests enriching dependent type theories with *higher-dimensional types*. Type theory is traditionally thought of as talking about sets, which are types of dimension 0: sets have elements (objects, or 0-cells), which may or may not be equal. Types of dimension 1 are like categories: they have elements, but also have a notion of map between elements (morphisms, or 1-cells), and every function on such a type must respect this notion of map. A setoid—a set equipped with an equivalence relation—is an example of a 1-dimensional type, but in general the notion of morphism can have computational content. An example is a category of sets with isomorphisms as morphisms—in this case, there can be many distinguishable maps between two sets. The type of contexts and structural properties is 1-dimensional as well. Types of dimension 2 have elements, maps, and maps between maps (2-cells)—an example is the collection of all categories, with functors as maps and natural transformations as maps between maps. This is an example of a *2-category*. In general, an n-dimensional type corresponds to an *n-category*, or, equivalently, a homotopy n-type.

One important reason to identify the dimension of a type is that different notions of equality are appropriate for types of different dimensions. For sets (0-types), there is nothing but equality of elements. For categories (1-types), elements can be equal, but they might also be *isomorphic*, if there are maps (1-cells) between them that compose to the identity. For categories, the

appropriate notion of equality is *equivalence*: maps back and forth, whose compositions are isomorphic to the identity, using the 2-cells (two categories are equivalent if there are functors back and forth that are naturally isomorphic to the identity). The goal of higher-dimensional type theory is to provide support for n-equivalence in full generality: n-equivalent terms should be provably equal in the theory, and all families of types and objects should respect n-equivalence.

On the category-theoretic and homotopy-theoretic side, this correspondence suggests using a higher-order type theory as a meta-language, both for category theory and homotopy theory themselves, and through them for all formalized mathematics—as has recently been espoused by Voevodsky (Voevodsky, 2010). In such a theory, mathematicians will always be able to reason up to equivalence, no matter the appropriate notion of equivalence for the objects in question.

The next question is what higher-dimensional type theory should look like. Dependent type theories distinguish two different notions of equality: *definitional equality* ($M \equiv N : A$), which is a judgement, and *propositional equality* ($\mathsf{Id}_A\ M\ N$), which is a type. Semantically, the right interpretation is that definitional equality means equality, but propositional equality means n-equivalence—and we will sometimes use this terminology for the two. In Martin-Löf's *extensional* type theory, definitional and propositional equality are identified, and all proofs of propositional equality are trivial; this is expressed by the *equality reflection* and *uniqueness of identity proofs* rules.

$$\frac{P : \mathsf{Id}_A\ M\ N}{M \equiv N : A}\ \textit{reflect} \qquad \frac{P : \mathsf{Id}_A\ M\ N}{P \equiv \mathsf{refl}_M : A}\ \textit{uip}$$

Note that the second rule requires the first to type-check. These rules are not sound for the aforementioned interpretation, as they say that equivalent objects are equal, and all equivalences are the identity. This is true for sets (0-types), but not for higher dimensions.

What is perhaps surprising is that Martin-Löf's *intensional* type theory is in fact sound for higher-dimensional types: there is nothing in intensional type theory that restricts the dimension of a type. Instead of equality reflection, propositional equality is eliminated by subst

$$\frac{x{:}A \vdash C\ \mathsf{type} \quad P : \mathsf{Id}_A\ M\ N \quad Q : C[M/x]}{\mathsf{subst}_C\ P\ Q : C[N/x]}$$

(or, more generally, the J rule, which allows a motive C that is dependent on the equivalence P as well). This rule says that any dependent type respects equivalence, which is true in any dimension. The lack of UIP has sometimes been seen as a problem, addressed by axioms such as Streicher's K, which is equivalent to asserting uniqueness of identity proofs propositionally:

```
uip-prop : (x y : A) (p q : Id A x y) -> Id (Id A x y) p q
```

or by making all equality proofs definitionally equal, as in OTT (Altenkirch et al., 2007). However, it can now be seen as a feature, which allows intensional type theory to be interpreted in higher-dimensional structures.

Higher-dimensional interpretations were pioneered in Hofmann and Streicher's groupoid interpretation (Hofmann and Streicher, 1998), which interprets intensional type theory in the 2-category of groupoids, functors, and natural transformations: A closed type denotes a groupoid, whose objects are the terms of the type, and whose morphisms are the (1-)equivalences between them. Identity and composition ensure that propositional equality is reflexive and transitive, while the groupoid structure, which demands that every map be invertible, ensures symmetry.

The hom-sets of a type A are presented in the syntax as the inhabitants of the identity type $\mathsf{Id}_A\ M\ N$. Open types and terms are interpreted as functors, whose object parts are (roughly) the usual sets and elements of the set-theoretic semantics, and whose morphism parts show how those types and terms preserve equivalence. When the context is not a discrete groupoid, the functorial action on equivalences is computationally relevant. For example, consider a universe set of sets and isomorphisms, by taking the propositional equalities between S_1 and S_2 to be invertible functions $\mathsf{El}(S_1) \to \mathsf{El}(S_2)$. The rule subst states that all type families respect isomorphism: for any $x : \mathsf{set} \vdash C : \mathsf{set}$, $A \cong B$ implies $C[A] \cong C[B]$. Computationally, the lifting of the isomorphism is given by the functorial action of the type family C.

Recent work has generalized this to higher dimensions. On the categorical side, Garner (2009) generalizes the groupoid interpretation to a class of 2-categories, rather than just $Groupoid$. Lumsdaine (2009) and van den Berg and Garner (2010) show that the syntax of intensional type theory forms a weak ω-category. On the homotopy-theoretic side, Awodey and Warren (2009) show how to interpret intensional type theory into abstract homotopy theory (Quillen model categories).

However, while intensional type theory is *sound* for higher dimensions, it is definitely incomplete: First, there is no way to introduce an identity type by writing down a higher equivalence. This is a shame, as every type constructor has an action on equivalences, but this action cannot be exploited in the syntax. Second, there is no way to observe the dimension of types—e.g. certain types that do satisfy UIP cannot be proved to do so. Voevodsky's univalence axiom (Voevodsky, 2010) is one way to rectify this. While theoretically sufficient, the axiomatic approach ignores the *definitional* behavior of the functorial action of each type constructor, and thus, there is still some work to be done to understand a syntactic treatment of higher-dimensional type theory.

7.1.4 Directed Type Theory

Intensional type theory is inherently groupoidal, in that propositional equality is provably symmetric. However, in our motivating application of programming with logics, the category Ctx is not a groupoid: weakening and substitution are asymmetric. This motivates the study of *directed dependent type theory*, which relaxes the symmetry requirement on proofs of equivalence to obtain a directed notion of *transformation* between elements of a type. Directedness has many other applications, as well: First, it extends the correspondence between type theory, ω-groupoids, and homotopy theory to directed type theory, ω-categories, and directed homotopy theory. A directed type theory would provide a more general meta-language for higher category theory and homotopy theory, allowing the expression of non-symmetric notions. While directed homotopy theory is not as well studied, it may have applications in physics, due to the directed aspects of space-time. On a more practical level, programming language semantics is full of directed notions—reduction, monotone functions on domains—and directed type theory may have applications in mechanization of these languages. It may also shed light on the problem of combining dependent types with concurrency, which has been analyzed in homotopy-theoretic terms (Gaucher, 2003).

Another reason to study directed type theory is that, as George Mallory would say, the familiar dependent type constructors have an action on directed transformations, and with a few simple changes to the theory we can expose this nature. Relaxing symmetry requires two main

changes to the type theory: First, the type theory makes the *variances* of type families explicit, so that we have the language to say, e.g., that Π is contravariant in its domain, but covariant in its range. Second, the type theory exposes higher-dimensional structure at the judgemental level, not through a propositional equality type. The reason for this is that a propositional equality type, in its standard formulation, relies subtly on the symmetry of equivalence. This comes up when proving that the type $\Gamma \vdash \mathsf{Id}_A\ M\ N\ \mathsf{type}$ is functorial in Γ. The crux of the matter is implementing the following function:

$$idfunc : \mathsf{Id}\ M_1\ M_2 \to \mathsf{Id}\ N_1\ N_2 \to \mathsf{Id}\ M_1\ N_1 \to \mathsf{Id}\ M_2\ N_2$$
$$idfunc = \lambda(a, b, c).b \circ c \circ a^{-1}$$

This is possible for equivalence, using inverses. However, given a *transformation* $M_1 \Longrightarrow M_2$ this function is not implementable. One solution to this problem, which we discuss below, is to investigate *directed hom types* that relate two terms of opposite variance. However, a more fundamental approach is to express the structure we want—transformations between two terms of the same variance—as a judgemental notion, which is not required to be functorial.

This judgemental approach to equality is arguably a better presentation even for symmetric type theory: The standard definition of the identity type is incomplete even for sets, as it does not allow one to prove that two functions are equal if they agree on all arguments (and similarly for other negative types). This can be patched by adding functional extensionality as an axiom (Hofmann, 1995), or by defining the identity type in a type-directed manner (Altenkirch et al., 2007). In either case, the definition of $\mathsf{Id}_A\ M\ N$ is no longer parametric in the type A. Thus, the rules for the identity type violate the principle that all propositional connectives should be independent. The common way to restore orthogonality is to analyze the offending type constructor as a judgement, not as a type. Additionally, the judgemental approach offers more freedom than the approach taken in OTT, where identity is a type defined in terms of other types: there is no reason to insist that the 2-cells be definable as 1-cells.

In this chapter, we take a first step in the investigation of directed type theory, considering the simplest non-trivial case, that of a 2-dimensional theory. *2-dimensional directed dependent type theory*, or *2DTT*, has three layers: types (0-cells), terms (1-cells), and transformations (2-cells), and can be interpreted in a 2-category. Each type itself denotes a 1-category, with terms as objects and transformations as morphisms. For simplicity, the syntax reflects the structure of a strict 2-category, where the associativity and unit laws of 1-cells are definitional equalities, not a weak 2-category (bicategory), where they hold only up to equivalence. Additionally, we treat *equality* (but not equivalence) as in extensional type theory—for example, we define a universe of extensional sets, with *reflect* and *uip*. We validate 2DTT by an interpretation in the strict 2-category Cat of categories, functors, and natural transformations, generalizing the groupoid interpretation—though we conjecture that this semantics should extend to a class of 2-categories with the appropriate structure.

In Section 7.2, we present the proof theory of 2DTT, followed by its semantics in Section 7.3.

Additional Related Work An early connection between λ-calculus and 2-categories was made by Seely (1987), who shows that simply-typed λ-calculus forms a (non-groupoidal) 2-category, with terms as 1-cells and reductions as 2-cells.

Variance annotations on variables are common in simply-typed subtyping systems (Cardelli, 1990; Duggan and Compagnoni, 1999; Steffen, 1998). In the dependently typed case, variance annotations have been used to support termination-checking using sized types, as in Mini-Agda (Abel, 2010).

Functoriality of simple and polymorphic type constructors has been studied in previous work on generic traversals of data structures (Bellè et al., 1996; Laemmel and Peyton Jones, 2003) and compilation of subtyping (Crary, 2000); our work generalizes this to the dependently typed case. In tactic-based proof assistants, it is possible to construct a library of tactics for showing that types and terms respect equivalence relations and order relations, such as Jackson's library for NuPRL (Jackson, 1996) and setoid rewriting in Coq (Coq Development Team, 2009). Our approach here is akin to building these tactics into the language, equipping *every* type and term with an action on transformations. This allows the computational content and equational behavior of these actions to be drawn out. Jackson (Jackson, 1995) also describes several examples of computationally relevant notions of equality that arise when formalizing algebra, such as permutations on lists, the associate relation in the factorization theory of integral domains, and the equality relation on a group quotiented by a computationally interesting normal subgroup. Porting these examples will be an interesting test of a higher-dimensional type theory.

7.2 Base Theory

In this section, we give a proof theory for 2DTT. But first, we discuss a few general methodological points:

Admissible versus derivable rules First, 2DTT requires several involution, identity, and composition (substitution) principles. These could potentially be made admissible and defined as meta-operations, but we have instead chosen to internalize them as syntactic forms and explain their meaning via definitional equality rules. For example, we make use of explicit substitutions, rather than treating substitution as a meta-level operation. That said, we leave weakening admissible, as the de Bruijn form that results from explicit weakening is difficult to read. The treatment of dependent types in Pitts (2000)'s survey article provides an introduction to this style of syntax, with an explicit substitution judgement and internalized composition principles. Presentationally, this style makes it easier to comprehend the entire theory. Semantically, it constrains the models of the theory to strict 2-categories, because these principles and equations must hold not just in the syntax, but in any extension of the theory.

That said, an alternative presentation using meta-operations will be a helpful guide to implementation, and our current formalism anticipates this alternative to some extent. In particular, we set out a methodology for defining types and contexts in 2DTT, which ensures that the involution, identity, and composition principles are defined (via definitional equality) in terms of other constructs in the theory. Because of this goal, some of the equations in the present theory are redundant, in that they can be derived from others, but these redundant equations would be necessary if these principles and their equations were made admissible.

Intrinsic versus extrinsic encodings Thus far in this thesis, we have made heavy use of *intrinsic encodings*, where programs are identified with typing derivations, and there are no raw/untyped programs. Such a presentation is undesirable for a syntactic presentation of a dependent type theory, because the type equality rule says that *the same term* has two different types:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv B\ \mathsf{type}}{\Gamma \vdash M : B}$$

If we identified terms with derivations, then this cast would show up in the syntax of a program, and therefore have to be reasoned about.

The right way of thinking about intrinsic encodings for dependent types is that the indices are not syntax, but *semantics*: The intrinsically typed set of terms $\mathsf{Tm}\ \Gamma\ A$ is indexed by a semantic type for which the equations proscribed by $\Gamma \vdash A \equiv B\ \mathsf{type}$ are *true*. This way, type conversion is inherited from the meta-language. One way to realize this is to view the syntax of the theory as an abstract specification of a structure with certain operations, e.g. in the style of *categories with families* (Dybjer, 1996). However, in this style of presentation the syntax is defined abstractly to be the least structure with these operations, rather than by a concrete inductive definition. An alternative is define the syntax inductively, but indexed by semantics, not syntax—$\mathsf{Tm}\ \Gamma\ A$ is defined inductively, but Γ and A are semantic entities (McBride, 2010). This requires defining the syntax mutually recursively with its interpretation, using induction-recursion (Dybjer, 2000).

Because we would like a concrete description of the syntax, and would like this presentation to be accessible to those without grounding in advanced techniques such as induction-recursion, we opt for a traditional extrinsic formulation. However, this decision forces us to confront some syntactic questions that we might have otherwise avoided: When the typing judgements are indexed by syntax, there is a question of whether the rules presuppose or guarantee that their subjects are well-formed. For example, in $\Gamma \vdash M : A$, A is syntactically a type, but is it well-formed in the sense of the judgement $\Gamma \vdash A\ \mathsf{type}$? There are several possible answers to this question:

1. Whenever we write $\Gamma \vdash M : A$, we will presuppose the existence of a separate derivation of $\Gamma \vdash A\ \mathsf{type}$. So the judgement $\Gamma \vdash M : A$ can (potentially) be formally derived when A is not well-formed, but we never consider such cases.
2. The subject of the judgement $\Gamma \vdash M : A$ is really a well-formed A: there is an extra argument $\mathcal{D} :: \Gamma \vdash A\,\mathsf{type}$ (perhaps treated proof-irrelevantly, so the identity of derivations does not matter).
3. The judgement $\Gamma \vdash M : A$ ensures that $\Gamma \vdash A\ \mathsf{type}$ holds, in the sense that the theorem "if $\Gamma \vdash M : A$ then $\Gamma \vdash A\ \mathsf{type}$" is provable by induction.
4. The judgement $\Gamma \vdash M : A$ ensures that $\Gamma \vdash A\ \mathsf{type}$ holds, in that $\Gamma \vdash A\ \mathsf{type}$ is directly derivable from the premises of each rule, without assuming an invariant about the system as a whole.
5. The judgement $\Gamma \vdash M : A$ ensures that $\Gamma \vdash A\ \mathsf{type}$ is a premise of the rule.

A related issue is which type annotations are present in the syntax of terms: are the domains of λ annotated? Is type information available at every level in the syntax tree?

When answering these questions, there is a tension between parsimony, convenience (of writ-

ing down terms), and implementation efficiency—generally pushing towards fewer premises and annotations—and the truth (or provability by a particular technique) of meta-theoretic properties on the other—generally pushing towards more. For example, it is sometimes the case that a premise is unnecessary, but this can only be seen after developing the meta-theory of the calculus.

In this work, our main meta-theoretic result is a soundness theorem interpreting the syntax in Cat. This theorem is akin to translating the extrinsic syntactic formulation into a (particular) intrinsic semantic formulation. Of course, not every syntactic term has a meaning—only the well-typed ones—so the interpretation is only total on well-typed terms. However, because the only non-syntax-directed rules are equality rules, which are interpreted as real semantic equality, the interpretation is *proof-irrelevant* in the derivation: the meaning of a term under any two derivations is the same. Indeed, it is helpful to know that this uniqueness property holds while proving the semantic interpretation total.

One way to achieve this is to first define a partial interpretation on raw terms, so that it is obviously independent of the derivation, and then prove this relation total on well-typed terms (Hofmann, 1995; Streicher, 1991). However, because the output of the semantics is an intrinsically typed notion, it must be possible to generate not just the meaning of a term, but its semantic type, from the term itself, independently of the typing derivation. This can be achieved by adopting the following stance: First, the context is treated as a presupposition in the sense of option 1 above: the rules never explicitly judge the context to be well-formed, but supply enough information to maintain this invariant. Second, every other meta-variable free in a rule must be an explicit argument to the proof-term; as a consequence, terms are explicitly annotated with type annotation at each level of the tree. For example, the function and application constructors for $\Pi\, x{:}A.\, B$ are $\lambda\, x{:}A.\, M^B$ and $\mathsf{app}_{A,x.B}(M, N)$. Second, each rule has a formation premise for each meta-variable appearing in the rule, such that under these premises the premise and conclusion judgements are well-formed (option 4 above). For example,

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma\,, x{:}A \vdash B\ \mathsf{type} \quad \Gamma \vdash M : \Sigma\, x{:}A.\, B}{\Gamma \vdash \mathsf{snd}_{A,x.B} M : B[\mathsf{fst}\ M/x]}$$

However, to make the syntax and rules more readable, we adopt an informal convention that these extra arguments and premises are suppressed. For example, we write the usual

$$\frac{\Gamma \vdash M : \Sigma\, x{:}A.\, B}{\Gamma \vdash \mathsf{snd}\ M : B[\mathsf{fst}\ M/x]}$$

to mean the above. Additionally, though every equation is a rule for a typed definitional equality judgement $\Gamma \vdash t = t' : J$, we elide the contexts, types, and formation premises, and present the rules as simple equations. The formation premises can be reconstructed by unrolling the typing rules of the left-hand side of the equation.

This syntax is more verbose than the standard natural deduction formulation—but, on the other hand, the standard natural deduction formulation is more verbose than necessary, as it fails to differentiate between situations where type information flows from the conclusion to the premises, and situations where it flows from the premises to the conclusion. A bidirectional presentation of 2DTT is an interesting subject for future work.

Involution

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma^{\mathsf{op}}\ \mathsf{ctx}} \quad \frac{\Gamma^{\mathsf{op}} \vdash \theta : \Delta^{\mathsf{op}}}{\Gamma \vdash \theta^{\mathsf{op}} : \Delta} \quad \frac{\Gamma^{\mathsf{op}} \vdash \delta : \theta'^{\mathsf{op}} \Longrightarrow_{\Delta^{\mathsf{op}}} \theta^{\mathsf{op}}}{\Gamma \vdash \delta^{\mathsf{op}} : \theta \Longrightarrow_{\Delta} \theta'}$$

$$\begin{array}{lcll} (\Gamma^{\mathsf{op}})^{\mathsf{op}} & \equiv & \Gamma & \textit{0-involution} \\ (\theta^{\mathsf{op}})^{\mathsf{op}} & \equiv & \theta & \textit{1-involution} \\ (\delta^{\mathsf{op}})^{\mathsf{op}} & \equiv & \delta & \textit{2-involution} \end{array}$$

Identity and composition for $\Gamma \vdash \theta : \Delta$

$$\frac{\Gamma \supseteq \Delta}{\Gamma \vdash \mathsf{id}_{\Delta} : \Delta} \quad \frac{\Gamma_2 \vdash \theta_2 : \Gamma_3 \quad \Gamma_1 \vdash \theta_1 : \Gamma_2}{\Gamma_1 \vdash \theta_2[\theta_1] : \Gamma_3} \quad \frac{\Gamma \vdash \theta : \Delta \quad \Gamma_0 \vdash \delta : \theta_1 \Longrightarrow_{\Gamma} \theta_2}{\Gamma_0 \vdash \theta[\delta] : \theta[\theta_1] \Longrightarrow_{\Delta} \theta[\theta_2]}$$

$$\begin{array}{lcll} \theta_0[\theta[\theta']] & \equiv & \theta_0[\theta][\theta'] & \textit{1-subst assoc/unit} \\ \theta_0[\mathsf{id}_{\Gamma}] & \equiv & \theta_0 & \\ \mathsf{id}^{\Gamma}_{\Gamma}[\theta] & \equiv & \theta & \end{array}$$

$$\begin{array}{lcll} \theta[\delta[\delta']] & \equiv & \theta[\delta][\delta'] & \textit{1-resp assoc} \\ \theta[\mathsf{refl}_{\theta'}] & \equiv & \mathsf{refl}_{\theta[\theta']} & \textit{1-resp preserves refl.} \\ \theta[\theta'][\delta] & \equiv & \theta[\theta'[\delta]] & \textit{1-resp for 1-subst} \end{array}$$

$$\begin{array}{lcll} \mathsf{id}^{\mathsf{op}}_{\Gamma} & \equiv & \mathsf{id}_{\Gamma^{\mathsf{op}}} & {}^{\mathsf{op}}\ \textit{interactions} \\ (\theta_1[\theta_2])^{\mathsf{op}} & \equiv & \theta_1^{\mathsf{op}}[\theta_2^{\mathsf{op}}] & \\ (\theta[\delta])^{\mathsf{op}} & \equiv & \theta^{\mathsf{op}}[\delta^{\mathsf{op}}] & \end{array}$$

Identity and Composition for $\Gamma \vdash \delta : \theta \Longrightarrow_{\Delta} \theta'$

$$\frac{}{\Gamma \vdash \mathsf{refl}^{\Delta}_{\theta} : \theta \Longrightarrow_{\Delta} \theta} \quad \frac{\Gamma \vdash \delta_1 : \theta_1 \Longrightarrow_{\Delta} \theta_2 \quad \Gamma \vdash \delta_2 : \theta_2 \Longrightarrow_{\Delta} \theta_3}{\Gamma \vdash \delta_2 \circ \delta_1 : \theta_1 \Longrightarrow_{\Delta} \theta_3} \quad \frac{\Gamma \vdash \delta : \theta \Longrightarrow_{\Delta} \theta' \quad \Gamma_0 \vdash \delta_0 : \theta_0 \Longrightarrow_{\Gamma} \theta'_0}{\Gamma_0 \vdash \delta[\delta_0] : \theta[\theta_0] \Longrightarrow_{\Delta} \theta'[\theta'_0]}$$

$$\begin{array}{lcll} (\delta_3 \circ \delta_2) \circ \delta_1 & \equiv & \delta_3 \circ (\delta_2 \circ \delta_1) & \textit{trans assoc/unit} \\ (\delta \circ \mathsf{refl}) & \equiv & \delta & \\ (\mathsf{refl} \circ \delta) & \equiv & \delta & \end{array}$$

$$\begin{array}{lcll} \delta_0[\delta[\delta']] & \equiv & \delta_0[\delta][\delta'] & \textit{2-resp assoc/unit} \\ \delta_0[\mathsf{refl}_{\mathsf{id}}] & \equiv & \delta_0 & \\ \mathsf{refl}_{\mathsf{id}^{\Gamma}_{\Gamma}}[\delta] & \equiv & \delta & \end{array}$$

$$\begin{array}{lcll} (\delta_1 \circ \delta_2)[\delta_3 \circ \delta_4] & \equiv & \delta_1[\delta_3] \circ \delta_2[\delta_4] & \textit{interchange} \\ \mathsf{refl}_{\theta}[\delta] & \equiv & \theta[\delta] & \textit{delegate} \end{array}$$

$$\begin{array}{lcll} \mathsf{refl}^{\mathsf{op}}_{\theta} & \equiv & \mathsf{refl}_{\theta^{\mathsf{op}}} & {}^{\mathsf{op}}\ \textit{interactions} \\ (\delta_1 \circ \delta_2)^{\mathsf{op}} & \equiv & \delta_2^{\mathsf{op}} \circ \delta_1^{\mathsf{op}} & \\ (\delta_1[\delta_2])^{\mathsf{op}} & \equiv & \delta_1^{\mathsf{op}}[\delta_2^{\mathsf{op}}] & \end{array}$$

Figure 7.1: 2DTT: Identity, Composition, and Involution Principles (1)

Composition for $\Gamma \vdash A\ \mathsf{type}$

$$\frac{\Gamma \vdash \theta : \Delta \quad \Delta \vdash A\ \mathsf{type}}{\Gamma \vdash A[\theta]\ \mathsf{type}} \qquad \frac{\Delta\ \mathsf{ctx} \quad \Delta \vdash C\ \mathsf{type} \quad \Gamma \vdash \delta : \theta_1 \Longrightarrow_\Delta \theta_2 \quad \Gamma \vdash M : C[\theta_1]}{\Gamma \vdash \mathsf{map}_{\Delta.C}\ \delta\ M : C[\theta_2]}$$

$$\begin{array}{rcll} A[\theta[\theta']] & \equiv & A[\theta][\theta'] & \textit{0-subst assoc/unit} \\ A[\mathsf{id}_\Gamma] & \equiv & A & \end{array}$$

$$\begin{array}{rcll} \mathsf{map}_{\Delta.C}\ \mathsf{refl}_\theta\ M & \equiv & M & \textit{0-resp functoriality} \\ \mathsf{map}_{\Delta.C}\ (\delta_2 \circ \delta_1)\ M & \equiv & \mathsf{map}_{\Delta.C}\ \delta_2\ (\mathsf{map}_{\Delta.C}\ \delta_1\ M) & \end{array}$$

$$\begin{array}{rcll} (\mathsf{map}_{\Delta.C}\ \delta\ M)[\theta_0] & \equiv & \mathsf{map}_{\Delta.C}\ \delta[\mathsf{refl}_{\theta_0}]\ M[\theta_0] & \textit{1-subst for } \mathsf{map} \\ (\mathsf{map}_C\ (\delta : \theta_1 \Longrightarrow \theta_2)\ M)[\delta' : \theta_1' \Longrightarrow \theta_2'] & \equiv & \mathsf{resp}^1\ (x.\mathsf{map}_C\ (\delta[\mathsf{refl}_{\theta_2'}])\ x)\ (M[\delta']) & \textit{1-resp for } \mathsf{map} \\ \mathsf{map}_{\Delta.C[\theta:\Delta']}\ \delta\ M & \equiv & \mathsf{map}_{\Delta'.C}\ \mathsf{refl}_\theta[\delta]\ M & \textit{def. } \mathsf{map} \textit{ for } A[\theta] \end{array}$$

Composition for $\Gamma \vdash M : A$

$$\frac{\Gamma \vdash \theta : \Delta \quad \Delta \vdash M : A}{\Gamma \vdash M[\theta] : A[\theta]} \qquad \frac{\Delta \vdash M : A \quad \Gamma \vdash \delta : \theta_1 \Longrightarrow_\Delta \theta_2}{\Gamma \vdash M[\delta] : (\mathsf{map}_{\Delta.A}\ \delta\ (M[\theta_1])) \Longrightarrow_{A[\theta_2]} M[\theta_2]}$$

$$\begin{array}{rcll} M[\theta[\theta']] & \equiv & M[\theta][\theta'] & \textit{1-subst assoc/unit} \\ M[\mathsf{id}_\Gamma] & \equiv & M & \end{array}$$

$$\begin{array}{rcll} M[\delta[\delta']] & \equiv & M[\delta][\delta'] & \textit{1-resp assoc/unit} \\ M[\mathsf{refl}_\theta] & \equiv & \mathsf{refl}_{M[\theta]} & \textit{1-resp preserves refl.} \\ M[\theta][\delta] & \equiv & M[\theta[\delta']] & \textit{1-resp for 1-subst} \end{array}$$

Identity and Composition for $\Gamma \vdash \alpha : M \Longrightarrow_A N$

$$\frac{}{\Gamma \vdash \mathsf{refl}^A_M : M \Longrightarrow_A M} \qquad \frac{\Gamma \vdash \alpha_1 : M_1 \Longrightarrow_A M_2 \quad \Gamma \vdash \alpha_2 : M_2 \Longrightarrow_A M_3}{\Gamma \vdash \alpha_2 \circ \alpha_1 : M_1 \Longrightarrow_A M_3} \qquad \frac{\Gamma_0 \vdash \delta_0 : \theta_0 \Longrightarrow_\Gamma \theta_0' \quad \Gamma \vdash \alpha : M \Longrightarrow_A N}{\Gamma_0 \vdash \alpha[\delta_0] : (\mathsf{map}_{\Gamma.A}\ \delta_0\ (M[\theta_0])) \Longrightarrow_{A[\theta_0']} N[\theta_0']}$$

$$\begin{array}{rcll} (\alpha_3 \circ \alpha_2) \circ \alpha_1 & \equiv & \alpha_3 \circ (\alpha_2 \circ \alpha_1) & \textit{trans assoc/unit} \\ (\alpha \circ \mathsf{refl}) & \equiv & \alpha & \\ (\mathsf{refl} \circ \alpha) & \equiv & \alpha & \end{array}$$

$$\begin{array}{rcll} \alpha[\delta[\delta']] & \equiv & \alpha[\delta][\delta'] & \textit{2-resp assoc/unit} \\ \alpha[\mathsf{refl}_{\mathsf{id}}] & \equiv & \alpha & \end{array}$$

$$\begin{array}{rcll} (\alpha_1 \circ \alpha_2)[\delta_3 \circ \delta_4] & \equiv & \alpha_1[\delta_3] \circ \mathsf{resp}^1\ (x.\mathsf{map}\ \delta_3\ x)\ (\alpha_2[\delta_4]) & \textit{interchange} \\ \mathsf{refl}_M[\delta] & \equiv & M[\delta] & \textit{delegate} \end{array}$$

Figure 7.2: 2DTT: Identity, Composition, and Involution Principles (2)

All judgements respect equality:

$$\frac{\Gamma \equiv \Gamma' \quad \Delta \equiv \Delta' \quad \Gamma' \vdash \theta : \Delta'}{\Gamma \vdash \theta : \Delta} \qquad \frac{\Gamma \equiv \Gamma' \quad \Gamma' \vdash A\ \mathsf{type}}{\Gamma \vdash A\ \mathsf{type}} \qquad \frac{\Gamma \equiv \Gamma' \quad \Gamma \vdash A \equiv A'\ \mathsf{type} \quad \Gamma' \vdash M : A'}{\Gamma \vdash M : A}$$

$$\frac{\Gamma \equiv \Gamma' \quad \Delta \equiv \Delta' \quad \Gamma \vdash \theta_1 \equiv \theta_1' : A \quad \Gamma \vdash \theta_2 \equiv \theta_2' : A \quad \Gamma' \vdash \delta : \theta_1' \Longrightarrow_{\Delta'} \theta_2'}{\Gamma \vdash \delta : \theta_1 \Longrightarrow_{\Delta} \theta_2}$$

$$\frac{\Gamma \equiv \Gamma' \quad \Gamma \vdash A \equiv A'\ \mathsf{type} \quad \Gamma \vdash M \equiv M' : A \quad \Gamma \vdash N \equiv N' : A \quad \Gamma' \vdash \alpha : M' \Longrightarrow_{A'} N'}{\Gamma \vdash \alpha : M \Longrightarrow_{A} N}$$

Equality respects equality: Each equality judgement has an analogous respect-for-equality rule, which says that it respects equality in the context and classifier.

Congruence: Each equality judgement is a congruence, specified by reflexivity, symmetry, transitivity rules, and a compatibility rule for each term constructor.

Figure 7.3: 2DTT: General equality rules

Empty context:

$$\frac{}{\cdot\ \mathsf{ctx}} \qquad \frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{}{\Gamma \vdash \cdot : \cdot \Longrightarrow_{\cdot} \cdot}$$

$$\begin{array}{lcll}
\theta & \equiv & \cdot & \textit{1-}\eta \\
\delta & \equiv & \cdot & \textit{2-}\eta \\
\cdot^{\mathsf{op}} & \equiv & \cdot & \textit{0,1,2-involution} \\
\mathsf{id}_{\cdot} & \equiv & \cdot & \textit{identity} \\
\cdot[\theta] & \equiv & \cdot & \textit{1-subst} \\
\cdot[\delta] & \equiv & \cdot & \textit{1-resp} \\
\mathsf{refl}_{\cdot} & \equiv & \cdot & \textit{reflexivity} \\
\cdot \circ \cdot & \equiv & \cdot & \textit{trans} \\
\cdot[\delta] & \equiv & \cdot & \textit{2-resp}
\end{array}$$

Covariant term variables:

$$\frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma\ ,\ x{:}A^{+}\ \mathsf{ctx}} \qquad \frac{x{:}A^{+} \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash \theta : \Delta \quad \Gamma \vdash M : A[\theta]}{\Gamma \vdash \theta, M^{+}/x : \Delta, x : A^{+}} \qquad \frac{\begin{array}{c}\Gamma \vdash \delta : \theta \Longrightarrow_{\Delta} \theta' \\ \Gamma \vdash \alpha : (\mathsf{map}_{\Delta.A}\ \delta\ M) \Longrightarrow_{A[\theta']} N\end{array}}{\Gamma \vdash (\delta, \alpha^{+}/x) : (\theta, M^{+}/x) \Longrightarrow_{\Delta, x:A^{+}} (\theta', N^{+}/x)}$$

$$\begin{array}{rcll}
\mathsf{id}^{\Gamma, x:A^{+}}_{\Gamma}[\theta, M^{+}/x] & \equiv & \theta & \textit{1-}\beta \\
x[\theta, M^{+}/x] & \equiv & M & \textit{1-}\beta \\
\theta : (\Gamma\ ,\ x{:}A^{+}) & \equiv & \mathsf{id}_{\Gamma}[\theta], x[\theta]^{+}/x & \textit{1-}\eta \\
\mathsf{id}^{\Gamma, x:A^{+}}_{\Gamma}[\delta, \alpha^{+}/x] & \equiv & \delta & \textit{2-}\beta \\
x[\delta, \alpha^{+}/x] & \equiv & \alpha & \textit{2-}\beta \\
\delta : \theta \Longrightarrow_{(\Gamma, x:A^{+})} \theta' & \equiv & \mathsf{id}_{\Gamma}[\delta], x[\delta]^{+}/x & \textit{2-}\eta \\
 & & & \\
\mathsf{id}_{\Gamma, x:A^{+}} & \equiv & \mathsf{id}_{\Gamma}, x^{+}/x & \textit{1-id} \\
(\theta, M^{+}/x)[\theta_0] & \equiv & \theta[\theta_0], M[\theta_0]^{+}/x & \textit{1-subst} \\
(\theta, M^{+}/x)[\delta_0] & \equiv & \theta[\delta_0], M[\delta_0]^{+}/x & \textit{1-resp} \\
\mathsf{refl}_{\theta, M^{+}/x} & \equiv & \mathsf{refl}_{\theta}, \mathsf{refl}_{M^{+}}/x & \textit{refl} \\
(\delta_2, {\alpha_2}^{+}/x) \circ (\delta_1, {\alpha_1}^{+}/x) & \equiv & (\delta_2 \circ \delta_1), (\alpha_2 \circ \mathsf{resp}^{1}\ (x.\mathsf{map}_{\Delta.A}\ \delta_2\ x)\ \alpha_1)^{+}/x & \textit{trans} \\
(\delta, \alpha^{+}/x)[\delta_0] & \equiv & \delta[\delta_0], \alpha[\delta_0]^{+}/x & \textit{2-resp} \\
 & & & \\
(\Gamma\ ,\ x{:}A^{+})^{\mathsf{op}} & \equiv & \Gamma^{\mathsf{op}}\ ,\ x{:}A^{-} & \textit{0-invol} \\
(\theta, M^{+}/x)^{\mathsf{op}} & \equiv & \theta^{\mathsf{op}}, M^{-}/x & \textit{1-invol} \\
(\delta, \alpha^{+}/x)^{\mathsf{op}} & \equiv & \delta^{\mathsf{op}}, \alpha^{-}/x & \textit{2-invol}
\end{array}$$

Figure 7.4: 2DTT: Contexts (1)

Contravariant term variables:

$$\frac{\Gamma\ \mathsf{ctx} \quad \Gamma^{\mathsf{op}} \vdash A\ \mathsf{type}}{\Gamma\ ,\ x{:}A^{\text{-}}\ \mathsf{ctx}} \qquad \frac{\Gamma \vdash \theta : \Delta \quad \Gamma^{\mathsf{op}} \vdash M : A[\theta^{\mathsf{op}}]}{\Gamma \vdash \theta, M^{\text{-}}/x : \Delta, x : A^{\text{-}}} \qquad \frac{\begin{array}{c}\Gamma \vdash \delta : \theta \Longrightarrow_{\Delta} \theta' \\ \Gamma^{\mathsf{op}} \vdash \alpha : (\mathsf{map}_{\Delta^{\mathsf{op}}.A}\ (\delta^{\mathsf{op}})\ N) \Longrightarrow_{A[\theta]} M\end{array}}{\Gamma \vdash (\delta, \alpha^{\text{-}}/x) : (\theta, M^{\text{-}}/x) \Longrightarrow_{\Delta, x:A^{\text{-}}} (\theta', N^{\text{-}}/x)}$$

$$\begin{array}{rcll}
\mathsf{id}_{\Gamma}^{\Gamma, x:A^{\text{-}}}[\theta, M^{\text{-}}/x] & \equiv & \theta & \textit{1-}\beta \\
\theta : (\Gamma\ ,\ x{:}A^{\text{-}}) & \equiv & \mathsf{id}_{\Gamma}[\theta], x[\theta^{\mathsf{op}}]^{\text{-}}/x & \textit{1-}\eta \\
\mathsf{id}_{\Gamma}^{\Gamma, x:A^{\text{-}}}[\delta, \alpha^{\text{-}}/x] & \equiv & \delta & \textit{2-}\beta \\
\delta : \theta \Longrightarrow_{(\Gamma, x:A^{\text{-}})} \theta' & \equiv & \mathsf{id}_{\Gamma}[\delta], x[\delta^{\mathsf{op}}]^{\text{-}}/x & \textit{2-}\eta \\
 & & & \\
\mathsf{id}_{\Gamma, x:A^{\text{-}}} & \equiv & \mathsf{id}_{\Gamma}, x^{\text{-}}/x & \textit{1-id} \\
(\theta, M^{\text{-}}/x)[\theta_0] & \equiv & \theta[\theta_0], M[\theta_0^{\mathsf{op}}]^{\text{-}}/x & \textit{1-subst} \\
(\theta, M^{\text{-}}/x)[\delta_0] & \equiv & \theta[\delta_0], M[\delta_0^{\mathsf{op}}]^{\text{-}}/x & \textit{1-resp} \\
\mathsf{refl}_{\theta, M^{\text{-}}/x} & \equiv & \mathsf{refl}_{\theta}, \mathsf{refl}_{M}{}^{\text{-}}/x & \textit{refl} \\
(\delta_2, \alpha_2{}^{\text{-}}/x) \circ (\delta_1, \alpha_1{}^{\text{-}}/x) & \equiv & (\delta_2 \circ \delta_1), (\alpha_2 \circ \mathsf{resp}^{1}\ (x.\mathsf{map}_{\Delta^{\mathsf{op}}.A}\ \delta_2^{\mathsf{op}}\ x)\ \alpha_1)^{\text{-}}/x & \textit{trans} \\
(\delta, \alpha^{\text{-}}/x)[\delta_0] & \equiv & \delta[\delta_0], \alpha[\delta_0^{\mathsf{op}}]^{\text{-}}/x & \textit{2-resp} \\
 & & & \\
(\Gamma\ ,\ x{:}A^{\text{-}})^{\mathsf{op}} & \equiv & \Gamma^{\mathsf{op}}\ ,\ x{:}A^{+} & \textit{0-invol} \\
(\theta, M^{\text{-}}/x)^{\mathsf{op}} & \equiv & \theta^{\mathsf{op}}, M^{+}/x & \textit{1-invol} \\
(\delta, \alpha^{\text{-}}/x)^{\mathsf{op}} & \equiv & \delta^{\mathsf{op}}, \alpha^{+}/x & \textit{2-invol}
\end{array}$$

Figure 7.5: 2DTT: Contexts (2)

Dependent functions:

$$\frac{\Gamma^{\mathsf{op}} \vdash A\ \mathsf{type} \quad \Gamma\,,\, x{:}A^{\text{-}} \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi\, x{:}A.\, B\ \mathsf{type}} \qquad \frac{\Gamma\,,\, x{:}A^{\text{-}} \vdash M : B}{\Gamma \vdash \lambda\, x.\, M : \Pi\, x{:}A.\, B} \qquad \frac{\Gamma \vdash M_1 : \Pi\, x{:}A.\, B \quad \Gamma^{\mathsf{op}} \vdash M_2 : A}{\Gamma \vdash M_1\ M_2 : B[M_2/x]}$$

$$\frac{\Gamma\,,\, x{:}A^{\text{-}} \vdash \alpha : (M\ x) \Longrightarrow_B (N\ x)}{\Gamma \vdash \lambda\, x.\, \alpha : M \Longrightarrow_{\Pi\, x:A.\, B} N} \qquad \frac{\Gamma \vdash \alpha : M \Longrightarrow_{\Pi\, x:A.\, B} N \quad \Gamma^{\mathsf{op}} \vdash \beta : N_1 \Longrightarrow_A M_1}{\Gamma \vdash \alpha\ \beta : \mathsf{map}^1_B\ \beta\ (M M_1) \Longrightarrow_{B[N_1/x]} (N N_1)}$$

$(\lambda\, x.\, M)\ N$	$\equiv$	$M[N^{\text{-}}/x]$	*1-β*
$M : \Pi\, x{:}A.\, B$	$\equiv$	$\lambda\, x.\, M\ x$	*1-η*
$(\lambda\, x.\, \alpha_1)\ \alpha_2$	$\equiv$	$\alpha_1[\mathsf{refl}, \alpha_2{}^{\text{-}}/x]$	*2-β*
$\alpha : M \Longrightarrow_{\Pi\, x:A.\, B} N$	$\equiv$	$\lambda\, x.\, \alpha\ (\mathsf{refl}_x)$	*2-η*
$(\Pi\, x{:}A.\, B)[\theta_0]$	$\equiv$	$\Pi\, x{:}A[\theta_0^{\mathsf{op}}].\, B[\theta_0, x^{\text{-}}/x]$	*0-subst*
$\mathsf{map}_{\Delta.\Pi\, x:A.\, B}\ \delta\ M$	$\equiv$	$\lambda\, x.\, \mathsf{map}_{\Delta, x:A^{\text{-}}.B}\ (\delta, \mathsf{refl})\ (M\ (\mathsf{map}_{\Delta^{\mathsf{op}}.A}\ \delta^{\mathsf{op}}\ x))$	*0-resp*
$(\lambda\, x.\, M)[\theta_0]$	$\equiv$	$\lambda\, x.\, M[(\theta_0, x^{\text{-}}/x)]$	*1-subst*
$(M_1\ M_2)[\theta_0]$	$\equiv$	$(M_1[\theta_0])\ (M_2[\theta_0])$	*1-subst*
$(\lambda\, x.\, M)[\delta]$	$\equiv$	$\lambda\, x.\, M[\delta, \mathsf{refl}]$	*1-resp*
$(M\ N)[\delta]$	$\equiv$	$M[\delta]\ N[\delta]$	*1-resp*
refl_M	$\equiv$	$\lambda\, x.\, \mathsf{refl}_{M\ x}$	*refl*
$(\lambda\, x.\, \alpha_2) \circ (\lambda\, x.\, \alpha_1)$	$\equiv$	$\lambda\, x.\, \alpha_2 \circ \alpha_1$	*trans*
$(\lambda\, x{:}A.\, \alpha)[\delta_0]$	$\equiv$	$\lambda\, x{:}A[\theta'].\, \alpha[\delta_0, \mathsf{refl}/a]$	*2-resp*
$\alpha_1\ \alpha_2[\delta_0]$	$\equiv$	$(\alpha_1[\delta_0])\ (\alpha_2[\delta_0])$	*2-resp*

Figure 7.6: 2DTT: Dependent Function Types

Dependent pairs:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x{:}A^{+} \vdash B\ \mathsf{type}}{\Gamma \vdash \Sigma\, x{:}A.\, B\ \mathsf{type}} \qquad \frac{\Gamma \vdash M_1 : A \quad \Gamma \vdash M_2 : B[M_1/x]}{\Gamma \vdash (M_1, M_2) : \Sigma\, x{:}A.\, B} \qquad \frac{\Gamma \vdash M : \Sigma\, x{:}A.\, B}{\Gamma \vdash \mathsf{fst}\ M : A} \qquad \frac{\Gamma \vdash M : \Sigma\, x{:}A.\, B}{\Gamma \vdash \mathsf{snd}\ M : B[\mathsf{fst}\ M/x]}$$

$$\frac{\Gamma \vdash \alpha_1 : \mathsf{fst}\ M \Longrightarrow_A \mathsf{fst}\ N \quad \Gamma \vdash \alpha_2 : (\mathsf{map}^1_B\ \alpha_1\ (\mathsf{snd}\ M)) \Longrightarrow_{B[\mathsf{fst}\ N/x]} \mathsf{snd}\ N}{\Gamma \vdash (\alpha_1, \alpha_2) : M \Longrightarrow_{\Sigma\, x:A.\, B} N} \qquad \frac{\Gamma \vdash \alpha : M \Longrightarrow_{\Sigma\, x:A.\, B} N}{\Gamma \vdash \mathsf{fst}\ \alpha : \mathsf{fst}\ M \Longrightarrow_A \mathsf{fst}\ N}$$

$$\frac{\Gamma \vdash \alpha : M \Longrightarrow_{\Sigma\, x:A.\, B} N}{\Gamma \vdash \mathsf{snd}\ \alpha : (\mathsf{map}^1_B\ (\mathsf{fst}\ \alpha)\ (\mathsf{snd}\ M)) \Longrightarrow_{B[\mathsf{fst}\ N/x]} \mathsf{snd}\ N}$$

$\mathsf{fst}\ (M, N)$	$\equiv$	M	*1-β*
$\mathsf{snd}\ (M, N)$	$\equiv$	N	*1-β*
$M : \Sigma\, x{:}A.\, B$	$\equiv$	$(\mathsf{fst}\ M, \mathsf{snd}\ M)$	*1-η*
$\mathsf{fst}\ (\alpha_1, \alpha_2)$	$\equiv$	α_1	*2-β*
$\mathsf{snd}\ (\alpha_1, \alpha_2)$	$\equiv$	α_2	*2-β*
$\alpha : M \Longrightarrow_{\Sigma\, x:A.\, B} N$	$\equiv$	$(\mathsf{fst}\ \alpha, \mathsf{snd}\ \alpha)$	*2-η*
$(\Sigma\, x{:}A.\, B)[\theta_0]$	$\equiv$	$\Sigma\, x{:}A[\theta_0].\, B[\theta_0, x^{+}/x]$	*0-subst*
$\mathsf{map}_{\Delta.\Sigma\, x:A.\, B}\ \delta\ M$	$\equiv$	$(\mathsf{map}_{\Delta.A}\ \delta\ (\mathsf{fst}\ M), \mathsf{map}_{\Delta, x:A^{+}.B}\ (\delta, \mathsf{refl}_{\mathsf{map}\ \delta\ (\mathsf{fst}\ M)})\ (\mathsf{snd}\ M))$	*0-resp*
$((M_1, M_2))[\theta_0]$	$\equiv$	$(M_1[\theta_0], M_2[\theta_0])$	*1-subst*
$(\mathsf{fst}\ M)[\theta_0]$	$\equiv$	$\mathsf{fst}\ (M[\theta_0])$	*1-subst*
$(\mathsf{snd}\ M)[\theta_0]$	$\equiv$	$\mathsf{snd}\ (M[\theta_0])$	*1-subst*
$(M, N)[\delta]$	$\equiv$	$(M[\delta], N[\delta])$	*1-resp*
$(\mathsf{fst}\ M)[\delta]$	$\equiv$	$\mathsf{fst}\ M[\delta]$	*1-resp*
$(\mathsf{snd}\ M)[\delta]$	$\equiv$	$\mathsf{snd}\ M[\delta]$	*1-resp*
refl_M	$\equiv$	$(\mathsf{refl}_{\mathsf{fst}\ M}, \mathsf{refl}_{\mathsf{snd}\ M})$	*refl*
$(\alpha_2, \alpha_2') \circ (\alpha_1, \alpha_1')$	$\equiv$	$(\alpha_2 \circ \alpha_1, \alpha_2' \circ \mathsf{resp}^1\ (x.\mathsf{map}_{\Delta.A}\ (\mathsf{refl}, \alpha_1)\ x)\ \alpha_1')$	*trans*
$(\alpha_1, \alpha_2)[\delta_0]$	$\equiv$	$(\alpha_1[\delta_0], \alpha_2[\delta_0, \mathsf{refl}])$	*2-resp*
$(\mathsf{fst}\ \alpha)[\delta_0]$	$\equiv$	$\mathsf{fst}\ \alpha[\delta_0]$	*2-resp*
$(\mathsf{snd}\ \alpha)[\delta_0]$	$\equiv$	$\mathsf{snd}\ \alpha[\delta_0]$	*2-resp*

Figure 7.7: 2DTT: Dependent Pairs

Sets and elements:

$$\frac{}{\Gamma \vdash \mathsf{set}\ \mathsf{type}} \qquad \frac{\Gamma \vdash S : \mathsf{set}}{\Gamma \vdash \mathcal{D}(S)\ \mathsf{type}} \qquad \frac{\Gamma\,,\, x{:}\mathcal{D}(S)^{+} \vdash M : \mathcal{D}(S')}{\Gamma \vdash x.M : S \Longrightarrow_{\mathsf{set}} S'} \qquad \frac{}{\Gamma \vdash \star : M \Longrightarrow_{\mathcal{D}(S)} M} \qquad \frac{\Gamma \vdash \alpha : M \Longrightarrow_{\mathcal{D}(S)} N}{\Gamma \vdash M \equiv N : \mathcal{D}(S)}$$

$$\begin{array}{rcll}
\mathsf{map}_{\mathcal{D}(S)}\ \delta\ M[\theta_1] & \equiv & M[\theta_2] & \textit{def. } \mathcal{D}(-) \\
\alpha : S \Longrightarrow_{\mathsf{set}} S' & \equiv & x.\mathsf{map}_{a:Set.\,\mathcal{D}(a)}\ (\cdot, \alpha)\ x & \textit{2-}\eta \\
\alpha : M \Longrightarrow_{\mathcal{D}(S)} N & \equiv & \star & \textit{2-}\eta \\
 & & & \\
\mathsf{set}[\theta_0] & \equiv & \mathsf{set} & \textit{0-subst} \\
(\mathcal{D}(S))[\theta_0] & \equiv & \mathcal{D}(S[\theta_0]) & \\
\mathsf{map}_{\Delta.\mathsf{set}}\ \delta\ M & \equiv & M & \textit{0-resp} \\
\mathsf{map}_{\Delta.\mathcal{D}(S)}\ \delta\ M & \equiv & N[\mathsf{id}, M^{+}/x] \ \text{ if } \ S[\delta] \equiv x.N & \\
\mathsf{refl}_{S\,:\,\mathsf{set}} & \equiv & x.x & \textit{refl} \\
\mathsf{refl}_{M\,:\,\mathcal{D}(S)} & \equiv & \star & \textit{refl} \\
(x.M_1) \circ (x.M_2) & \equiv & x.M_2[M_1/x] & \textit{trans} \\
\star \circ \star & \equiv & \star & \textit{trans} \\
(x.M)[\delta_0] & \equiv & x.\mathsf{map}_{\Delta.S'}\ \delta_0\ M[\theta, x^{+}/x] \text{ if } \Delta\,,\, x{:}\mathcal{D}(S)^{+} \vdash M : \mathcal{D}(S') \text{ and } \delta_0 : \theta \Longrightarrow \theta' & \textit{2-resp} \\
\star_M[\delta_0] & \equiv & M[\delta_0] & \textit{2-resp}
\end{array}$$

Figure 7.8: 2DTT: General Rules for Sets and Elements

$$\frac{\Gamma^{\mathsf{op}} \vdash S : \mathsf{set} \quad \Gamma\,,\, x{:}\mathcal{D}(S)^{-} \vdash S' : \mathsf{set}}{\Gamma \vdash \Pi\, x{:}S.\, S' : \mathsf{set}} \qquad \frac{\Gamma \vdash S : \mathsf{set} \quad \Gamma\,,\, x{:}\mathcal{D}(S)^{+} \vdash S' : \mathsf{set}}{\Gamma \vdash \Sigma\, x{:}S.\, S' : \mathsf{set}} \qquad \frac{}{\Gamma \vdash 0, 1, 2 : \mathsf{set}} \qquad \frac{\Gamma \vdash S : \mathsf{set} \quad \Gamma \vdash M, N : \mathcal{D}(A)}{\Gamma \vdash \mathsf{Id}_S\ M\ N : \mathsf{set}}$$

$$\frac{\Gamma \vdash M : \Sigma\, x{:}\mathcal{D}(S).\, \mathcal{D}(S')}{\Gamma \vdash \mathsf{in}\, M : \mathcal{D}(\Sigma\, x{:}S.\, S')} \qquad \frac{\Gamma \vdash M : \mathcal{D}(\Sigma\, x{:}S.\, S')}{\Gamma \vdash \mathsf{out}\, M : \Sigma\, x{:}\mathcal{D}(S).\, \mathcal{D}(S')}$$

$$\frac{\Gamma \vdash M : \Pi\, x{:}\mathcal{D}(S).\, \mathcal{D}(S')}{\Gamma \vdash \mathsf{in}\, M : \mathcal{D}(\Pi\, x{:}S.\, S')} \qquad \frac{\Gamma \vdash M : \mathcal{D}(\Pi\, x{:}S.\, S')}{\Gamma \vdash \mathsf{out}\, M : \Pi\, x{:}\mathcal{D}(S).\, \mathcal{D}(S')}$$

$$\frac{}{\Gamma \vdash () : \mathcal{D}(1)} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathcal{D}(2)} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathcal{D}(2)} \qquad \frac{\Gamma^{\pm} \vdash M : \mathcal{D}(0)}{\Gamma \vdash \mathsf{abort}\, M : C} \qquad \frac{\Gamma^{\pm} \vdash M : \mathcal{D}(0)}{\Gamma \vdash \mathsf{abort}\, M : M_1 \Longrightarrow_C M_2}$$

$$\frac{\begin{array}{l}\Gamma^{\pm} \vdash M : \mathcal{D}(2) \\ \Gamma \vdash M_1 : C[\mathsf{true}^{\pm}/x] \\ \Gamma\,,\, x{:}2^{\pm} \vdash C\ \mathsf{type} \\ \Gamma \vdash M_2 : C[\mathsf{false}^{\pm}/x]\end{array}}{\Gamma \vdash \mathsf{if}_{x^{\pm}.C}(M, M_1, M_2) : C[M^{\pm}/x]} \qquad \frac{\begin{array}{l}\Gamma^{\pm} \vdash M : \mathcal{D}(2) \\ \Gamma\,,\, x{:}2^{\pm} \vdash C\ \mathsf{type} \\ \Gamma\,,\, x{:}2^{\pm} \vdash M_1, M_2 : C \\ \Gamma \vdash \alpha_1 : M_1[\mathsf{true}^{\pm}/x] \Longrightarrow_{C[\mathsf{true}^{\pm}/x]} M_2[\mathsf{true}^{\pm}/x] \\ \Gamma \vdash \alpha_2 : M_1[\mathsf{false}^{\pm}/x] \Longrightarrow_{C[\mathsf{false}^{\pm}/x]} M_2[\mathsf{false}^{\pm}/x]\end{array}}{\Gamma \vdash \mathsf{if}_{x^{\pm}.C}(M, \alpha_1, \alpha_2) : M_1[M^{\pm}/x] \Longrightarrow_{C[M^{\pm}/x]} M_2[M^{\pm}/x]}$$

$$\frac{\Gamma \vdash \alpha : M \Longrightarrow_{\mathcal{D}(S)} N}{\Gamma \vdash \mathsf{idi}\, \alpha : \mathsf{Id}_S\ M\ N} \qquad \frac{\Gamma \vdash P : \mathsf{Id}_{\mathcal{D}(S)}\ M\ N}{\Gamma \vdash \mathsf{ide}\, P : M \Longrightarrow_{\mathcal{D}(S)} N}$$

Rules for sets:

$$\begin{array}{lcll}
\Pi\, x{:}S.\, S'[\theta] & \equiv & \Pi\, x{:}S[\theta].\, S'[\theta, x^{-}/x] & \textit{1-subst} \\
\Sigma\, x{:}S.\, S'[\theta] & \equiv & \Sigma\, x{:}S[\theta].\, S'[\theta, x^{+}/x] & \\
\{0,1,2\}[\theta] & \equiv & \{0,1,2\} & \\
\mathsf{Id}_S\ M\ N[\theta] & \equiv & \mathsf{Id}_{A[\theta]}\ M[\theta]\ N[\theta] & \\
\{0,1,2\}[\delta] & \equiv & x.x & \textit{1-resp} \\
(\Pi\, x{:}S.\, S')[\delta : \theta \Longrightarrow \theta'] & \equiv & x.\mathsf{in}\,(\mathsf{map}_{\Pi\, x:\mathcal{D}(S).\, \mathcal{D}(S')}\ \delta\ (\mathsf{out}\, M)) & \\
(\Sigma\, x{:}S.\, S')[\delta : \theta \Longrightarrow \theta'] & \equiv & x.\mathsf{in}\,(\mathsf{map}_{\Sigma\, x:\mathcal{D}(S).\, \mathcal{D}(S')}\ \delta\ (\mathsf{out}\, M)) & \\
\mathsf{Id}_S\ M\ N[\delta : \theta \Longrightarrow \theta'] & \equiv & x.\mathsf{idi}\, \star &
\end{array}$$

Figure 7.9: 2DTT: Some Sets (1)

Rules for elements:

$$
\begin{array}{lcll}
\mathsf{out}\,(\mathsf{in}\,M) & \equiv & M & \beta\eta \\
\mathsf{in}\,(\mathsf{out}\,M) & \equiv & M & \\
M : \mathcal{D}(1) & \equiv & () & \\
M[(N : \mathcal{D}(0))^{\pm}/x] & \equiv & \mathsf{abort}\,N & \\
\mathsf{if}\,(\mathsf{true}, M_1, M_2) & \equiv & M_1 & \\
\mathsf{if}\,(\mathsf{false}, M_1, M_2) & \equiv & M_2 & \\
M[(N : \mathcal{D}(2))^{\pm}/x] & \equiv & \mathsf{if}\,(N, M[\mathsf{true}^{\pm}/x], M[\mathsf{false}^{\pm}/x]) & \\
\mathsf{ide}\,(\mathsf{idi}\,\alpha) & \equiv & \alpha & \\
P : \mathsf{Id}_S\ M\ N & \equiv & \mathsf{idi}\,(\mathsf{ide}\,P) &
\end{array}
$$

$$
\begin{array}{lcll}
(\mathsf{in}\,M)[\theta] & \equiv & \mathsf{in}\,M[\theta] & \textit{1-subst} \\
(\mathsf{out}\,M)[\theta] & \equiv & \mathsf{out}\,M[\theta] & \\
\{(), \mathsf{true}, \mathsf{false}\}[\theta] & \equiv & \{(), \mathsf{true}, \mathsf{false}\} & \\
\mathsf{if}_{x\,:\,2^{\pm}.C}(M, M_1, M_2)[\theta^{\Gamma}_{\Delta}] & \equiv & \mathsf{if}_{x\,:\,2^{\pm}.C[\theta, x^{\pm}/x]}(M[\theta], M_1[\theta], M_2[\theta]) & \\
(\mathsf{idi}\,\alpha)[\theta] & \equiv & \mathsf{idi}\,\alpha[\mathsf{refl}_{\theta}] & \\
(\mathsf{ide}\,P)[\delta] & \equiv & \star & \textit{2-resp}
\end{array}
$$

$$
\begin{array}{lcll}
(\mathsf{in}\,M)[\delta] & \equiv & \star & \textit{1-resp} \\
(\mathsf{out}\,M)[\delta] & \equiv & \star & \\
\{(), \mathsf{true}, \mathsf{false}\}[\delta] & \equiv & \star & \\
\mathsf{idi}\,\alpha[\delta] & \equiv & \mathsf{idi}\,\star & \\
\mathsf{if}\,(M, M_1, M_2)[\delta : \theta_1 \Longrightarrow \theta_2] & \equiv & \mathsf{if}\,(M[\theta_2], M_1[\delta], M_2[\delta]) &
\end{array}
$$

Rules for transformation elims for positives:

$$
\begin{array}{rcll}
\mathsf{if}_{x.C}(\mathsf{true}, \alpha_1, \alpha_2) & \equiv & \alpha_1 & \beta \\
\mathsf{if}_{x.C}(\mathsf{false}, \alpha_1, \alpha_2) & \equiv & \alpha_2 & \\
\alpha[(\mathsf{refl}_{M\,:\,0})^{\pm}/x] & \equiv & \mathsf{abort}\,M & \eta \\
\alpha[(\mathsf{refl}_{M\,:\,2})^{\pm}/x] & \equiv & \mathsf{if}\,(M, \alpha[\mathsf{refl}_{\mathsf{true}}/x], \alpha[\mathsf{refl}_{\mathsf{false}}/x]) &
\end{array}
$$

$$
\begin{array}{rcll}
(\mathsf{abort}\,M)[\delta : \theta_1 \Longrightarrow \theta_2] & \equiv & \mathsf{abort}\,M[\theta_2] & \textit{2-resp} \\
\mathsf{if}\,(M, \alpha_1, \alpha_2)[\delta : \theta_1 \Longrightarrow \theta_2] & \equiv & \mathsf{if}\,(M[\theta_2], \alpha_1[\delta], \alpha_2[\delta]) &
\end{array}
$$

Figure 7.10: 2DTT: Some Sets (2)

7.2.1 Two-dimensional Judgements

2DTT has three main judgements, defining contexts Γ, substitutions θ, and transformations δ. In the semantics given below, these are interpreted as categories (0-cells), functors (1-cells), and natural transformations (2-cells), respectively. Each of these three levels has a corresponding contextualized version, which is judged well-formed relative to a context Γ. Contextualized contexts and substitutions are dependent types A and terms M, while contextualized transformations are transformations between terms. These judgements have the following form:

- Contexts: Γ ctx
- Substitutions: $\Gamma \vdash \theta : \Delta$ (where Γ ctx and Δ ctx)
- Transformations: $\Gamma \vdash \delta : \theta \Longrightarrow_{\Delta} \theta'$ (where Γ ctx and Δ ctx and $\Gamma \vdash \theta, \theta' : \Delta$)
- Dependent Types: $\Gamma \vdash A$ type (where Γ ctx)
- Terms: $\Gamma \vdash M : A$ (where Γ ctx and $\Gamma \vdash A$ type)
- Term Transformations: $\Gamma \vdash \alpha : M \Longrightarrow_A M'$ (where Γ ctx and $\Gamma \vdash A$ type and $\Gamma \vdash M, M' : A$)

7.2.2 Involution, Identity, and Composition Principles

In Figures 7.1 and 7.2, we present the generic involution, identity, and composition principles that define the basic structure of the theory. These principles and equations provide a contract that all specific contexts and types must satisfy.

Involution The involution rules say that there is a dualizing operation $^{\mathsf{op}}$ on contexts, substitutions, and transformations. The equations say that this dualization operation is involutive (self-inverse). The rule for θ^{op} says that the opposite of a substitution proves the opposite of the contexts. To avoid specializing the context in the conclusion of the rules, we phrase this as an "elimination" rule, removing $^{\mathsf{op}}$ from the two premise contexts. However, because $^{\mathsf{op}}$ is an involution, the following "introduction" rule is derivable:

$$\frac{\Gamma \vdash \theta : \Delta}{\Gamma^{\mathsf{op}} \vdash \theta^{\mathsf{op}} : \Delta^{\mathsf{op}}}$$

The dual of a transformation not only dualizes the contexts and substitutions, but also reverses the direction of the transformation.

Identity and Composition for Substitutions The next three rules define identity and composition for substitutions.

To make weakening admissible, id is really the composition of the identity substitution with *projections* that forget any number of variables (by our convention about implicit formation premises, there is a tacit premise that Δ is still well-formed). We write $\Gamma \supseteq \Delta$ to mean that Δ is obtained from Γ by dropping some number of variables. For convenience, we specify weakenings here for all the contexts we will consider here, though we could associate the clauses of

weakening with each form of context:

$$\frac{}{\Gamma \supseteq \cdot}\ \textit{done} \qquad \frac{\Gamma \supseteq \Gamma'}{\Gamma\,,\, x{:}A^{\pm} \supseteq \Gamma'}\ \textit{skip} \qquad \frac{\Gamma \supseteq \Gamma'}{\Gamma\,,\, x{:}A^{\pm} \supseteq \Gamma'\,,\, x{:}A^{\pm}}\ \textit{keep}$$

We do not require a rule for $^{\mathsf{op}}$ because $^{\mathsf{op}}$ can always be expanded away using equalities.

Composition of substitutions $\theta_2[\theta_2]$, which we refer to as 1-substitution, is standard in explicit substitution calculi. The additional composition operation, $\theta[\delta]$, forces substitutions to *respect transformation*: substitution instances by transformable substitutions are transformable. For this reason, we refer to it as *1-resp(ect)*. The first three equations say that 1-substitution is associative and unital. In the second equation, id_Γ can in fact be a weakening, in which case θ is tacitly weakened in the right-hand side. The third equation only makes sense when $\Gamma \vdash \mathsf{id} : \Gamma$, which we notate by $\mathsf{id}^\Gamma_\Gamma$. The next two rules say that 1-resp associates with 2-resp ($\delta[\delta']$), which is analogous operation for transformations (defined below), and preserves identities refl(defined below). The next three rules say that $^{\mathsf{op}}$ preserves identities (and projections), and distributes over compositions. As will often be the case, the second rule (for 1-cells) is necessary to typecheck the third (for 2-cells).

Because of explicit substitutions, 2DTT exploits a slightly interesting notion of variable binding: as is standard in explicit substitution calculi, the substitution operation $\theta_2[\theta_1]$ is a binding operator, with the range Γ_2 of θ_1 bound in θ_2—and similarly $\theta[\delta]$ and other composition principles. Here, we will assume we are working in an informal logical framework that supports this concept. It will be made precise using de Bruijn indices in the categorical semantics.

Identity and Composition For Transformations The next three rules define identity and composition for transformations. Transformations are always reflexive (refl) and transitive ($\delta_2 \circ \delta_1$). Additionally, transformations themselves respect transformation ($\delta[\delta_0]$), which we call 2-resp. The equations say that: Transitivity is associative and unital with reflexivity. 2-resp is also associative and unital. However, the unit of 2-resp is not an arbitrary refl_θ—which would still require adapting a transformation $\delta : \theta \Longrightarrow \theta'$ to $\theta[\theta_0] \Longrightarrow \theta[\theta_0]$—but only $\Gamma \vdash \mathsf{refl}_{\mathsf{id}^\Gamma_\Gamma} : \mathsf{id}_\Gamma \Longrightarrow_\Gamma \mathsf{id}_\Gamma$ (by above, $\theta[\mathsf{id}_\Gamma]$ equals θ). As above, the second rule holds when id is in fact a projection, but the third requires that it really be the identity. The *interchange law* relates 2-resp and transitivity: transitivity followed by 2-resp is the same as 2-resp followed by transitivity. It has a variety of useful special cases:

$$\begin{array}{rcll}
\theta[\delta \circ \delta'] & \equiv & \theta[\delta] \circ \theta[\delta'] & \textit{1-resp preserves transitivities} \\
(\delta \circ \delta')[\mathsf{refl}_\theta] & \equiv & \delta[\mathsf{refl}_\theta] \circ \delta'[\mathsf{refl}_\theta] & \textit{2-resp preserves transitivities} \\
(\delta : \theta_1 \Longrightarrow \theta_2)[\delta' : \theta'_1 \Longrightarrow \theta'_2] & \equiv & \mathsf{refl}_{\theta_2}[\delta'] \circ \delta[\mathsf{refl}_{\theta'_1}] & \textit{2-resp interchange 1} \\
(\delta : \theta_1 \Longrightarrow \theta_2)[\delta' : \theta'_1 \Longrightarrow \theta'_2] & \equiv & \delta[\mathsf{refl}_{\theta'_2}] \circ \mathsf{refl}_{\theta_1}[\delta'] & \textit{2-resp interchange 2}
\end{array}$$

The first two equations say that resp preserves transitivities. The next two state that a 2-resp is equivalent to holding one part fixed while doing one transformation, then holding the other fixed while doing the other—in either order. Returning to the figure, the rule *delegate* delegates 2-resp at reflexivity to 1-resp. The final three rules say that $^{\mathsf{op}}$ preserves identities and compositions, reversing the order of composition in the case of transitivity.

We do not define 2-subst, $\delta[\theta]$, directly, as this composition is definable as $\delta[\mathsf{refl}_\theta]$. Alternatively, we could take $\delta[\theta]$ as primitive and define $\delta[\delta']$ using the interchange law. However, it is in fact no harder to define $\delta[\delta']$, as the rules for the binary version also proceed compositionally in the term, just like ordinary substitution with a single θ would. This fact suggests that it may be possible to treat 2-resp as a meta-operation, which may be a helpful implementation technique.

Dependent Types In Figure 7.2, we tell the analogous story for dependent types, terms, and term transformations. A dependent type A can be pre-composed with a substitution, written $A[\theta]$; and has a functorial action $\mathsf{map}_{\Delta.A}\ \delta\ M$, which is the analogue of the subst elimination rule for propositional equality described above. map says that a transformation $\theta_1 \Longrightarrow \theta_2$ allows a term of type $A[\theta_1]$ to be coerced to a term of type $A[\theta_2]$. This says that *dependent types respect transformation*. We refer to these as *0-subst* and *0-resp*; there are no 0-subst/resp for contexts because contexts are not dependent.

The equations say: Substitution into types (0-subst) is associative and unital. map is functorial, preserving reflexivity and transitivity (this could alternatively be phrased as an interchange law, but the present formulation is simpler because we have not defined a *type* transformation judgement $A \Longrightarrow_{\mathsf{type}} B$). The next two rules define 1-subst and 1-resp for map, which reassociate the 1-subst/1-resp with the 0-resp. The next rule defines map for a composition, again by reassociating.

Interactions between map and 2-resp are derivable from the above equations for transitivity, using the interchange law:

$$\begin{array}{rcl} \mathsf{map}_C\ (\delta : \theta_1 \Longrightarrow \theta_2)[\delta' : \theta_1' \Longrightarrow \theta_2']\ M & \equiv & \mathsf{map}_C\ (\delta[\mathsf{refl}_{\theta_2'}])\ (\mathsf{map}_C\ \mathsf{refl}_{\theta_1}[\delta']\ M) \\ \mathsf{map}_C\ (\delta : \theta_1 \Longrightarrow \theta_2)[\delta' : \theta_1' \Longrightarrow \theta_2']\ M & \equiv & \mathsf{map}_C\ (\mathsf{refl}_{\theta_2}[\delta])\ (\mathsf{map}_C\ \delta[\mathsf{refl}_{\theta_1'}]\ M) \end{array}$$

The interchange rule, along with the following, form the Godement calculus of functors and natural transformation composition (Godement, 1958):

$$(\mathsf{refl}_{\theta[\theta']})[\delta] \equiv \mathsf{refl}_\theta[\mathsf{refl}_{\theta'}[\delta]]$$

This rule is derivable because $\mathsf{refl}_{\theta\circ\theta'} \equiv \mathsf{refl}_\theta[\mathsf{refl}'_\theta]$ by *1-resp-preserves-refl* and associativity.

There is also a right unit law for *resp* and $\mathsf{refl}_{\mathsf{id}}$:

$$\mathsf{refl}_\theta[\mathsf{refl}_{\mathsf{id}}] \equiv \mathsf{refl}_\theta$$

It is derivable using *1-resp preserves refl.* and unit of id.

Terms Like all contextual judgements, terms are closed under substitution ($M[\theta]$) and respect transformation ($M[\delta]$). Because terms are dependent on the context, the latter requires "adjusting" $M[\theta_1]$ by δ so that it lives in the same type as $M[\theta_2]$. The equality rules are analogous to those for substitutions: 1-subst is associative and unital, and 1-resp is associative and preserves reflexivities.

An associativity rule for 1-subst followed by 1-resp in a term is derivable, using 1-resp-preserves-refl, congruence, and delegate and associativity for 2-resp (see below):

$$M[\theta][\delta] \equiv M[\mathsf{refl}_\theta[\delta]] \quad \textit{1-resp for } M[\theta]$$

Using interchange, we can derive that M preserves transitivities from the above interaction with 2-resp:

$$\mathsf{refl}_M[\delta \circ \delta'] \equiv \mathsf{refl}_M[\delta] \circ (\mathsf{resp}^1\ (x.\mathsf{map}\ \delta\ x)\ (\mathsf{refl}_M[\delta']))$$

Term Transformations The rules for term transformations are entirely analogous to the rules for transformations, specifying reflexivity, transitivity, and 2-resp. The equations say that transitivity is associative and unital, that 2-resp is associative and unital, and that the order of trans and 2-resp can be interchanged. The interchange rule uses the derived form resp^1 , which is explained below.

General equality rules Figure 7.3 collects a variety of general equality rules: Each judgement, including equality, respects equality of its indices, and is a congruence.

7.2.3 Contexts

With the basic setup in hand, we are ready to define some concrete context formers. The general methodology for defining a context is to specify

1. A formation rule for Γ
2. A substitution rule $\theta : \Gamma$, and a hypothesis rule for one of the other judgements (e.g. the term rule for x for the context former $\Gamma , x{:}A^+$). These function as the introduction and elimination rules for the context, though we require contexts to be products of some sort, eliminated by first projections (which are implicit in id) and variables (representing projections).
3. A transformation rule for $\delta : \theta \Longrightarrow_\Gamma \theta'$
4. Equations defining

1-$\beta\eta$	$\beta\eta$ for θ
2-$\beta\eta$	$\beta\eta$ for δ
0-involution	Γ^{op}
1-involution	θ^{op}
2-involution	δ^{op}
identity	id_Γ
1-subst	$\theta[\theta']$
1-resp	$\theta[\delta']$
reflexivity	refl_θ
transitivity	$\delta \circ \delta'$
2-resp	$\delta[\delta']$

In general, refl and $\delta \circ \delta'$ are defined in a type-directed manner, by giving one rule that covers arbitrary arguments. On the other hand, the subst/resp principles are defined in a syntax-directed manner, giving one rule for each syntactic construct.

It may seem redundant that we ask that each context to give equations for both id_θ and $\theta[\delta]$, as $\mathsf{id}_\theta \equiv \mathsf{id}_\theta[\mathsf{refl}_{\mathsf{id}}] \equiv \theta[\mathsf{refl}_{\mathsf{id}}]$—so perhaps one operation could be derived from the other. However, because these definitions are mutually referentially, and we have not yet sorted out whether there is some induction order that allows one to be defined in terms of the other, we give the rules for both operations to be safe.

In Figure 7.4 and Figure 7.5 we carry out this methodology for the basic contexts:

Empty context The empty context has a trivial substitution into it, and a trivial transformation from this substitution to itself. Since the substitutions and transformations are singletons, the equations are all trivial.

Covariant context extensions Next, we present the rules for covariant context extension: if A is a type well-formed in Γ, then Γ can be extended with a variable of type A. A covariant variable can be used as a term; the typing rule checks that the variable is in the context:

$$\frac{}{x : A^+ \in (\Gamma, A^+)} \quad \frac{x : A^+ \in \Gamma}{x : A^+ \in (\Gamma\ ,\ y{:}B^\pm)}$$

As with weakening, we do not include a rule for $^{\mathsf{op}}$, which can be expanded away. The substitution into an extended context $\Delta\ ,\ x{:}A^+$ is a pair of a substitution θ into Δ and a term of type A, adjusted by θ (this is analogous to the usual introduction rule for a Σ-type). A transformation between such substitutions is a pair of transformations, one between the substitutions, and the other between the terms (adjusted by the first component). As these substitutions and transformations are pairs, the first set of rules gives the expected $\beta\eta$ rules, for the projections given by idand variables. The next rules define the identity, composition, and involution operations componentwise. The involution rules turn covariant context extension into contravariant context extension, which is defined below.

In the definition of refl, trans, and 2-resp, we cheat a little bit by assuming that the substitutions/transformations are in the form of the intro rules, when in fact these operations must be defined for arbitrary elements. This cheat is justified by the η-rules, and make the rules somewhat easier to read. Similarly, in the β-rules, we cheat by assuming that the substitution is in introductory form for exactly the context in question; this can always be achieved by reassociating. We also cheat a bit by not defining separate syntactic constructs for projections from context transformations, using $\mathsf{refl}_{\mathsf{id}}$ and refl_x instead—e.g. we should define $\mathsf{reflv}(x) : x \Longrightarrow_A x$, distinct from refl_x. These would be necessary for refl to be entirely definable in terms of other syntactic constructs.

The familiar resp congruence rule derives respect for the last variable in the context:

$$\frac{\Gamma \vdash \alpha : M \Longrightarrow_A N \quad \Gamma \vdash B\ \mathsf{type} \quad \Gamma\ ,\ x{:}A^+ \vdash F : B}{\Gamma \vdash \mathsf{resp}^1\ F\ \alpha : F[M/x] \Longrightarrow_B F[N/x]}$$

by $\mathsf{resp}^1\ F\ \alpha = F[\mathsf{id}_{\mathsf{id}}, \alpha^+/x]$. This is well-typed because map id cancels.

We will also make use of the corresponding rule for map, a derived form for transforming the last variable in the context:

$$\frac{\Gamma, x{:}A^+ \vdash B\ \mathsf{type} \quad \Gamma \vdash \alpha : M_1 \Longrightarrow_A M_2 \quad \Gamma \vdash M : B[{M_1}^+/x]}{\Gamma \vdash \mathsf{map}^1_{x:A^+.B}\ \alpha\ M : B[{M_2}^+/x]}$$

This is defined by $\mathsf{map}^1_{x:A^+.B}\ \alpha\ M = \mathsf{map}_{\Gamma,x:A^+.B}\ \mathsf{id}, \alpha^+/x\ M$.

Contravariant context extensions Next, we present the rules for contravariant context extension: if A is a type well-formed in Γ^{op}, then Γ can be extended with a variable of type A. For the most part, these are renamings of the rules for covariant context extension, except for the following: First, there is no rule for using a contravariant variable. This is because we reduce contravariant terms to covariant terms using $^{\mathsf{op}}$, and using the rules for $\Gamma\,, x{:}A^{-\mathsf{op}}$, a contravariant variable becomes a covariant variable. Second, the transformation rule reverses the order of M and N in the premise. Third, various $^{\mathsf{op}}$'s are inserted on substitutions and transformations to make the types work out.

A contravariant last-variable map rule is also definable:

$$\frac{\Gamma, x{:}A^- \vdash B\ \mathsf{type} \quad \Gamma^{\mathsf{op}} \vdash \alpha : M_2 \Longrightarrow_A M_1 \quad \Gamma \vdash M : B[{M_1}^-/x]}{\Gamma \vdash \mathsf{map}^1_{x:A^-.B}\ \alpha\ M : B[{M_2}^-/x]}$$

by $\mathsf{map}^1_{x:A^-.B}\ \alpha\ M = \mathsf{map}_{\Gamma,x:A^-.B}\ (\mathsf{id}, \alpha^-/x)\ M$.

7.2.4 Types

With contexts in hand, we can move on to types and terms. In general, a type is specified by:

1. A formation rule for A
2. Introduction and elimination term rules, defining $M : A$
3. Introduction and elimination transformation rules, defining $\alpha : M \Longrightarrow_A M'$
4. Equations defining

1-$\beta\eta$	$\beta\eta$ for M
2-$\beta\eta$	$\beta\eta$ for α
0-substitution	$A[\theta]$
0-resp	$\mathsf{map}_A\ \delta\ M$
1-substitution	$M[\theta]$
1-resp	$M[\delta]$
reflexivity	refl_M
transitivity	$\alpha \circ \alpha'$
2-resp	$\alpha[\delta]$

Dependent functions In Figure 7.6, we give the rules for dependent functions. The formation rule is standard, except that the domain is well-formed contravariantly in Γ, and thus assumed as a contravariant assumption. The intro and elim rules then insert the appropriate $^{\mathsf{op}}$'s. The transformation introduction rule says that a transformation at Π can be introduced by giving a family of transformations that work for each element—the extensionality rule. A transformation is eliminated by applying to transformable arguments. The symmetric variants of these transformation

rules, and the equations for them described below, have been considered in prior categorically-motivated accounts of functionally extensional propositional equality (Garner, 2009).

The $\beta\eta$-rules are the expected rules for functions, both at the term and transformation levels. We write $M[N/x]$ to abbreviate $M[\mathsf{id}, N/x]$. Substitution into a Π-type proceeds compositionally, though we always $^{\mathsf{op}}$ the substitution in contravariant positions. $\mathsf{map}_{\Pi\, x:A.\, B}$ is given by pre- and post-composition. Note that the definition of transformation at $\Delta, x : A^-$ is just right so that δ, refl works as the post-composition. 1-subst and 1-resp are both defined compositionally, as is 2-resp. The rule for refl says that the identity at all elements is the identity. In the definition of transitivity, we again cheat by assuming the transformations are in introductory form, which makes sense because of η, but we could equivalently use the elimination rule instead.

Dependent pairs The rules for Σ-types are mostly unsurprising, essentially a contextualized version of the rules for covariant context extension. The formation rule is analogous to Π, but the first component is well-formed covariantly, and the typing of the second component uses covariant context extension. The term rules are standard; the transformation rules say that a transformation between a pair is a pair of transformations. map is defined componentwise; in this case, the definition of transformation at $\Delta, x : A^+$ is just right so that δ, refl works as the second component. The $\beta\eta$-rules are standard, and the substitution rules are all defined compositionally. Identity and composition are defined componentwise.

Sets and elements As our first example of a base type with non-trivial transformations, we consider a universe set that contains *discrete types*. That is, each term $S : \mathsf{set}$ will represent a type $\mathcal{D}(S)$ whose elements have no non-identity transformations between them. However, set itself is *not* a discrete type: we take a transformation from S to S' to be a function from $\mathcal{D}(S)$ to $\mathcal{D}(S')$. Consequently, any type $s : \mathsf{set} \vdash C\ \mathsf{type}$ will admit a lifting of a function from $\mathcal{D}(S)$ to $\mathcal{D}(S')$ to a transformation from $C[S/s]$ to $C[S'/s]$. This is the common functor interface used in many programming languages. This universe of sets is *extensional*, in that transformation at sets satisfies equality reflection and definitional uniqueness of identity proofs—one can work with these sets as one would work in extensional type theory.

There are different ways of populating the types $\mathcal{D}(S)$ determined by a universe: First, one may define $\mathcal{D}(S)$ by recursion on S, computing an existing type that represents its elements. Second, one may define elements of the universe by giving a signature of constants and equations—using the type theory in the style of Martin-Löf's logical framework (Nordström et al., 1990). Third, one may give inference rules directly defining the members of these types. In what follows, we opt for the third option. There are two reasons for this choice. The first, which is somewhat technical, is discussed below—the set- and type-versions of Π and Σ are semantically isomorphic, but not equal—so defining $\mathcal{D}(S)$ by recursion would ignore this isomorphism. The second reason is that it is more convenient to specify a set, than a type, because the transformations between elements are always only reflexivity. A set is specified by:

1. A formation rule for $S : \mathsf{set}$, with equations for

1-substitution for S	$S[\theta]$
1-resp for S	$S[\delta]$

2. Terms defining $M : \mathcal{D}(S)$, as well as equations defining

$\beta\eta$	$\beta\eta$-rules for $\mathcal{D}(S)$
1-substitution	$M[\theta]$
1-resp	$M[\delta]$

For simplicity, we introduce sets by inference rules, though the signature approach would work as well.

The equations for 1-resp for each S define the functorial action of the set former—which is used by map. The equations for 1-resp for each term M are often computationally trivial, and involve only the verification of equations—any transformation at $\mathcal{D}(S)$ is reflexivity. However, we include these equations because the act of seeing that the trivial equation is well-typed is where one verifies that the appropriate equations hold.

Sets and Elements. In Figure 7.8, we present the generic rules that apply to all sets: set is a type, as is $\mathcal{D}(S)$ if S has type set. A transformation at set is a function from the elements of one to the elements of the other. The only transformation between elements of sets is reflexivity, and such transformations are eliminated by equality reflection.

The first equation says that 1-resp action of elements of sets is an equality, which is true because $\mathcal{D}(S)$ is a discrete type: if $M : \mathcal{D}(S)$ then M takes transformable arguments to *equal* results. The next two equations η-expand a transformation at set into a map, and a transformation at $\mathcal{D}(S)$ into reflexivity. The rules for 0-subst are compositional.

map at set is a no-op, because set is a constant functor. map at $\mathcal{D}(S)$ applies the function (open term) given by 1-resp of S, $S[\delta]$. We will give rules for each set-former defining $S[\delta]$. There are a couple of different ways in which $S[\delta]$ can be equal to a transformation $x.N$. The first is by η-expansion using *2-η*, but this expansion itself uses map, so the *0-resp* equation is trivial. The second, and more useful, is by applying composition (*1-resp*) and computation rules (*2-β*) until a transformation $x.N$ that does not simply re-introduce the map is computed. In the present calculus, a transformation at set can always be put into this form. If, however, we introduced transformation variables (see below), then $S[\delta]$ might compute to a variable transformation, in which case $\mathsf{map}_{\Delta.\mathcal{D}(S)}\ \delta\ M$ would not compute any further. In this case, it would be useful to add a separate elimination construct for applying a transformation at set to a term, in order to preserve the property that map can be defined in terms of other operations in the calculus.

Reflexivity and transitivity are defined in the expected way for sets, and trivially for elements. The 2-resp rule for set says that a function $x.M$ respects a transformation δ by running M at the source and then applying δ. The 2-resp rule for $\mathcal{D}(-)$ is analogous to *delegate*.

Basic Sets: Rules. In Figures 7.9 and 7.10, we present the rules for some basic sets: Π, Σ, 0 (the empty set), 1 (the unit set), 2 (booleans), and Id (identity between two elements of a set). Though we leave a general transformation type internalizing $\Rightarrow$ to future work, it is possible to define an identity type for symmetric types, which every discrete type $\mathcal{D}(S)$ trivially is.

The rules for Π and Σ express that they are isomorphic to Π/Σ types of discrete elements. We cheat a little here and break orthogonality of the connectives, defining Π and Σ by these isomorphisms (in fact, giving rules for $\mathcal{D}(S)$ at all breaks orthogonality, so perhaps we should have a separate judgement for the elements of a set)—otherwise we would have to spell out

functions, application, pairing and projections again. In fact, the domain of a Π need not be restricted to a set: $\Pi\, x{:}A.\, S$ is a set even if A is higher-dimensional. However, in our present theory types are both higher-dimensional than sets, and of higher *size* than sets: set itself is a type. So such a quantifier would not only be higher-dimensional, but impredicative. If additionally we had a universe typ of *small* types, then it would be appropriate to allow Π's to range over $A : \mathsf{typ}$.

$0, 1, 2$ are introduced by the usual rules. The elimination rules for 0 and 2 have one subtlety: they allow elimination of both co- and contravariantly well-formed terms—we abbreviate a choice between Γ and Γ^{op} by $\Gamma^{\pm}$. The reason for this is that the natural deduction rules for positive types build in a cut, and the appropriate notion of cut includes both co- and contra-variant cut formulas (cf. the fact that substitutions allow for both co- and contravariant variables). The negative types that we have considered thus far do not exhibit this phenomenon because their elimination rules do not build in a cut. Positives can also be eliminated towards transformation judgements, so we add rules for eliminating $\mathcal{D}(0)$ and $\mathcal{D}(2)$ towards $M \Longrightarrow_A N$. We could also add elimination rules towards other judgements (types, substitutions, transformations between substitutions), but what we have were is enough to illustrate the idea.

The rules for the identity type express an isomorphism with the corresponding term transformations.

Basic Sets: Equalities. 1-subst into each set is as expected. 1-resp is the identity for constant sets, defined in terms of Π and Σ types for Π and Σ, and computationally trivial for identity (the verification that the right-hand-side is well-typed uses symmetry of equality). Next, in Figure 7.10, the $\beta\eta$ rules express the isomorphisms for Π and Σ and Id, and the usual equations for 0,1,2. For symmetry, we include the coproduct rules for 0 and 2, but it they can also be proved using the transformation elimination rules and then reflected. 1-subst (and 2-resp for ide) are defined compositionally. The 1-resp rules are computationally trivial, but the fact that they are well-typed is interesting: e.g. the rule for in M requires showing that if two terms are transformable at $\Sigma\, x{:}\mathcal{D}(S).\, \mathcal{D}(S')$ then they are equal—which is true because the transformation provides equalities of each component. The rule for if uses the transformation elimination form for booleans. The transformation elims satisfy similar $\beta\eta$ and *2-resp* laws.

7.2.5 Weakening

Weakening is admissible for all judgements of the form $\Gamma \vdash J$:

LEMMA 7.2.1. *Weakening. If* $\Gamma \vdash J$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash J$.

Additionally, because id_Γ includes weakening, the equations allow pruning irrelevant components from a substitution. For example:

$$t[\theta, M^{\pm}/x] \equiv t[\theta] \text{ if } \Delta \vdash t : J \text{ and } \Gamma \vdash \theta : \Delta$$
$$t[\delta, \alpha^{\pm}/x] \equiv t[\delta] \text{ similarly}$$
$$\mathsf{map}_{\Delta, x : A^{\pm}.C}\ (\delta, \alpha^{\pm}/x)\ M \equiv \mathsf{map}_{\Delta.C}\ \delta\ M \text{ if } \Delta \vdash C\ \mathsf{type}$$

For the first, by unit $t \equiv t[\mathsf{id}]_\Gamma$; so by associativity $t[\theta, M^{\pm}/x] \equiv t[\mathsf{id}_\Gamma[\theta, M^{\pm}/x]]$, which β-reduces to $t[\theta]$. The remaining equations are similar, because $\mathsf{refl}_{\mathsf{id}}$ acts as a projection on transformations.

As an extremal case, if a term is closed, then the substitution/map is trivial. For example:

$$\mathsf{map}_{\Delta.C}\ \delta\ M \quad \equiv \quad M \quad \text{if} \quad \Delta \# C \quad \textit{def.}\ \mathsf{map}\ \textit{for constant}$$

$C \equiv C[\mathsf{id}.]$, so reassociating using *def.* map *for* $A[\theta]$ turns the transformation into $\mathsf{id}.[\delta]$, which is equal to the identity.

These observations allow us to derive the usual lookup rules for variables:

$$\begin{array}{rcll} x[\theta] & \equiv & \theta(x) & \textit{1-comp for variables} \\ x[\delta] & \equiv & \delta(x) & \textit{2-comp for variables} \end{array}$$

where we write $\theta(x)$ for the function that extracts the M/x component of θ, and similarly for δ. This is derivable by first projecting away the irrelevant components and then using the β rule—$x[\theta, M^+/x]$ and similarly for δ.

7.3 Semantics

In this section, we give a semantics in Cat, the 2-category of categories, functors, and natural transformations.

The intuition for this interpretation is that a context, or a closed type, is interpreted as a category, whose objects are the members of the type, and whose morphisms are the transformations between members. Thus, a substitution (an "open object") is interpreted as a functor—a family of objects that preserves transformations. A transformation (an "open morphism") is interpreted as a natural transformation—a family of morphisms that respects substitution. More formally, the context, substitution, and transformation judgements are interpreted as follows:

- $[\![\Gamma]\!]$ is a category
- $[\![\Gamma \vdash \theta : \Delta]\!]$ is a functor $[\![\theta]\!] : [\![\Gamma]\!] \longrightarrow [\![\Delta]\!]$
- $[\![\Gamma \vdash \delta : \theta_1 \Longrightarrow_\Delta \theta_2]\!]$ is a natural transformation $[\![\delta]\!] : [\![\theta_1]\!] \Longrightarrow [\![\theta_2]\!] : [\![\Gamma]\!] \longrightarrow [\![\Delta]\!]$

Next, we consider the semantics of types, terms, and term transformations.

Semantic Types The judgement $\Gamma \vdash A$ type represents an open type. Correspondingly, it should be interpreted as a functor that assigns a closed type to each object of Γ, preserving transformations. Since closed types are represented by categories, this is modeled by a functor into Cat:

$$[\![\Gamma \vdash A\ \mathsf{type}]\!] \text{ is a functor } [\![A]\!] : [\![\Gamma]\!] \longrightarrow Cat$$

To interpret the type set, we require a category of sets in Cat. Thus, we take Cat to be the category of large categories, so that it includes $Sets$ as an object. Because Cat is larger than $[\![\Gamma]\!]$, the space of functors $\Gamma \longrightarrow Cat$ relies on an implicit inclusion of large categories into "larger" categories.

As a notational convention, we will overload notation so that the semantics looks just like the syntax: First, we use the same letter for a piece of syntax and for the semantic concept it is interpreted as; e.g. we will write Γ for a category, θ for a functor, A for a functor into Cat, etc.

Second, we use the same symbols as we use in the syntax for the 2-category structure on Cat: we write the identity functor as id, functor composition as $\theta[\theta']$, vertical composition of natural transformations as $\delta \circ \delta'$, horizontal composition as $\delta[\delta']$, and the identity natural transformation as refl_θ, and the action of $^{\mathsf{op}}$ as Γ^{op}, etc. Additionally, we abbreviate $\Gamma \longrightarrow Cat$ by $\mathsf{Ty}\ \Gamma$.

Semantic Terms Above, we defined the category of elements of a functor $A : \Gamma \longrightarrow Sets$, $\int_\Gamma A$, which represents the context extension $\Gamma\,, x{:}\mathcal{D}(A)^+$. The general form of the Grothendieck construction works for a functor $A : \Gamma \to Cat$:

- an object of $\int_{\mathcal{C}} A$ is a pair (o, a) where $o \in \mathsf{Ob}\ \mathcal{C}$, and $a \in \mathsf{Ob}\ A(o)$.
- a morphism from (o, a) to (o', a') is a pair (c, f) where $c : o \longrightarrow_{\mathcal{C}} o'$ and $f : A(c)o \longrightarrow_{A(o')} o'$
- $\mathsf{id}_{(o,a)} = (\mathsf{id}_o, \mathsf{id}_a)$
- $(c, f) \circ (c', f') = (c \circ c', f \circ A(c)f')$

This differs from the definition given above by requiring a morphism for the second component, rather than an equality.

The Grothendieck construction constructs not only a category, but a (covariant) *fibration*.[1] Suppose $\mathsf{p} : E \longrightarrow \Gamma$. The fiber of E over $o \in \mathsf{Ob}\ \Gamma$ is the subcategory E_o consisting of those objects e of E such that $\mathsf{p}(e) = o$, and those morphisms f such $\mathsf{p}(f) = \mathsf{id}_o$. Then, modulo some technical details that we do not need to discuss here, p is a fibration iff pullback along a morphism $c : o_1 \longrightarrow o_2$ in Γ determines a functor between fibers E_{o_1} and E_{o_2}. As this definition suggests, there is an equivalence of categories between $Fib(\Gamma)$ (the fibrations over Γ) and Cat^Γ (the functor category). On objects, the right-to-left direction is the Grothendieck construction, and the left-to-right direction extracts a functor from a fibration—this extraction is not possible for an arbitrary functor $\mathsf{p} : E \longrightarrow \Gamma$, but the fibration condition ensures that it is. We will refer to the fibration functor $\mathsf{p} : \int_\Gamma A \longrightarrow \Gamma$ constructed by the Grothendieck construction as a *weakening map*; concretely, it is defined by first project on objects and morphisms.

The set of terms $\Gamma \vdash M : A$ is isomorphic to the one-element substitutions $\Gamma \vdash \mathsf{id}, M^+/x : \Gamma\,, x{:}A^+$, and $\Gamma\,, x{:}A^+$ is interpreted as $\int_\Gamma A$. Thus, we can define a the interpretation of a term $\Gamma \vdash M : A$ to be a functor $[\![M]\!] : \Gamma \longrightarrow \int_{[\![\Gamma]\!]} [\![A]\!]$ such that the Γ part of the functor is the identity—which we can formalize by saying that $[\![M]\!]$ is a section of p: $\mathsf{p} \circ [\![M]\!] = \mathsf{id}$. However, following Hofmann and Streicher (1998), it is more convenient to use an equivalent explicit definition:

DEFINITION 7.3.1. For a category Γ and a functor $A : \Gamma \longrightarrow Cat$, the set of terms over Γ of type A, written $\mathsf{Tm}\ \Gamma\ A$, consists of pairs (M_o, M_a) such that

- For all $\gamma \in \mathsf{Ob}\ \Gamma$, $M_o(\gamma) \in \mathsf{Ob}\ (A(\gamma))$
- For all $c : \gamma_1 \longrightarrow_\Gamma \gamma_2$, $M_a(c) : A(c)(M_o(\gamma_1)) \longrightarrow_{A(\gamma_2)} M_o(\gamma_2)$. Moreover, $M_a(\mathsf{id}) = \mathsf{id}$ and $M_a(c_2 \circ c_1) = M_a(c_2) \circ A(c)(M_a(c_1))$.

A $\mathsf{Tm}\ \Gamma\ A$ is a "dependently typed functor." When A is a constant functor, the definition reduces to that of a functor $M : \Gamma \longrightarrow A$. However, in general, the type of the object that M_o returns

[1] This is usually called an *opfibration*, as the word fibration is used for contravariant fibrations, which we will discuss below.

depends on its argument, and the action on morphisms relates $M_o(\gamma_2)$ to $(M_o(\gamma_1))$ "adjusted" by applying the functor $A(c)$. As with functors, we elide the projections from M, writing $M(\gamma)$ and $M(c)$.

Semantic Term Transformations Next, we define the semantic counterpart of $\Gamma \vdash \alpha : M \Longrightarrow_A N$. We characterized terms M as one-element substitutions, or sections of weakening. Similarly, we can define a transformation between M and N to be a transformation $(\mathsf{id}, M) \Longrightarrow (\mathsf{id}, N) : \Gamma \longrightarrow \int_\Gamma A$ that projects to the identity transformation on id. However, it is more convenient to work with the following equivalent definition:

DEFINITION 7.3.2. Given a category Γ, $A : \mathsf{Ty}\,\Gamma$, and $M, N : \mathsf{Tm}\,\Gamma\,A$, a *dependent natural transformation* $\alpha : M \Longrightarrow N$ consists of a family of maps α_γ such that

- for $\gamma \in \mathsf{Ob}\,\Gamma$, $\alpha_\gamma : M(\gamma) \longrightarrow_{A(\gamma)} N(\gamma)$
- for $c : \gamma_1 \longrightarrow_\Gamma \gamma_2$, $N(c) \circ A(c)(\alpha_{\gamma_1}) = \alpha_{\gamma_2} \circ M(c)$

Structure of the Interpretation Overall, our goal is to show that every well-formed context denotes a category, every substitution a functor, and so on. This can be factored into two parts:

The interesting part of the interpretation is showing that each inference rule is true, in the following sense: given the semantic domains corresponding to the premises, we can construct the semantic domain corresponding to the conclusion. For example, given a rule such as

$$\frac{\Delta \vdash M : A \quad \Gamma \vdash \theta : \Delta}{\Gamma \vdash M[\theta] : A[\theta]}$$

its semantic counterpart is a function

$$-[-] : (M : \mathsf{Tm}\;\Delta\;A) \to (\theta : \Gamma \longrightarrow \Delta) \to \mathsf{Tm}\;\Gamma\;A[\theta]$$

Once we have defined the operations, we can validate each equation on the semantic counterparts of the terms in question. Taken together, these constructions and proofs represent the inductive steps of the interpretation.

The second part of the interpretation is an induction on syntax and derivations that interprets each term as the corresponding semantic construction, and definitional equality as equations in the semantics. This step runs into some technical considerations, which we discuss below.

7.3.1 Semantic Judgemental Framework

First, we validate the rules and equations in Figures 7.1 and 7.2.

Involution The involutions are interpreted by the 2-functor $-^{\mathsf{op}} : Cat \longrightarrow Cat^{\mathsf{co}}$ which sends each category to its opposite category. $\Gamma^{\mathsf{op}}$is the action on objects; $\theta^{\mathsf{op}}$is the action on 1-cells; and $\delta^{\mathsf{op}}$is the action on 2-cells. More concretely, for any category Γ, $\Gamma^{\mathsf{op}}$is the category whose objects are the objects of Γ, and whose morphisms $o_1 \longrightarrow o_2$ are morphisms $o_2 \longrightarrow o_1$ in Γ. θ^{op} simply "retypes" a functor $C \longrightarrow D$ to a functor $C^{\mathsf{op}} \longrightarrow D^{\mathsf{op}}$, without changing the data involved. δ^{op} similarly retypes a natural transformation. $^{\mathsf{op}}$ is self-dual up to equality, which justifies the equations.

Identity and Composition for θ The identity and composition principles for substitutions and transformations are interpreted as the identity, horizontal composition, and vertical composition operations of the 2-category Cat. The equations are part of the definition of a 2-category. Composition of substitutions is interpreted as functor composition $\theta[\theta']$. $\theta[\delta]$ is interpreted as the horizontal composition $\mathsf{refl}_\theta[\delta]$—"whiskering" a natural transformation with a functor. Because id builds in weakening, it cannot be interpreted directly as the identity functor. Instead, the interpretation depends on the weakening, and uses the functors defined with each form of context below: *skip* is interpreted as the functor $\cdot : \Gamma \longrightarrow 1$. Given $\theta : \Gamma \longrightarrow \Gamma'$, *skip* is interpreted as the functor $\theta \circ \mathsf{p}^\pm : \Gamma.A^\pm \longrightarrow \Gamma$. *keep* is interpreted as the functor $(\theta \circ \mathsf{p}^\pm, \mathsf{v}^\pm) : \Gamma.A[\theta^\pm]^\pm \longrightarrow \Gamma.A$. When $\Gamma \vdash \mathsf{id} : \Gamma$, the interpretation is an η-expanded identity functor.

To validate the *1-subst assoc/unit* equations, we observe that functor composition is associative and unital, which justifies the first and third equations. The second equation requires weakening coherence (described below), as the left-hand side is interpreted with the weakening all the way at the outside, whereas the right-hand side is interpreted with it at the leaves. The *1-resp* equations are part of the definition of a 2-category. The $^{\mathsf{op}}$ interactions are part of 2-functoriality of $^{\mathsf{op}}$.

Identity and Composition for δ refl_θ is the identity natural transformation; $\delta_2 \circ \delta_1$ is composition of natural transformations, or *vertical composition* in the 2-category Cat. $\delta[\delta_0]$ is horizontal composition of natural transformations. The *assoc/unit* equations state associativity and unit of these compositions, which hold in any 2-category. The second *2-resp* rule depends on weakening coherence, like the corresponding 1-subst rule. The interchange law also holds in any 2-category, and we defined $\theta[\delta]$ to make *delegate* true. The $^{\mathsf{op}}$ interactions are the rest of 2-functoriality of $^{\mathsf{op}}$.

Types A type is interpreted as a functor, and $A[\theta]$ as functor composition.

map is an instance of whiskering a functor (into Cat) with a natural transformation, which can be thought of as the functorial action of the type on the transformation. For reference, we spell out the explicit construction: given a type $C : \mathsf{Ty}\ \Gamma$, a natural transformation $\delta : \theta \Longrightarrow \theta' : \Gamma \longrightarrow \Delta$, and a term $M : \mathsf{Tm}\ \Gamma\ C[\theta]$, we define a term $\mathsf{Tm}\ \Gamma\ C[\theta']$ as follows:

$$(\mathsf{map}_C\ \delta\ M)(\sigma) = C(\delta_\sigma)(M(\sigma))$$
$$(\mathsf{map}_C\ \delta\ M)(c : \sigma_1 \longrightarrow_\Gamma \sigma_2) = C(\delta_{\sigma_2})(M(c))$$

The action on morphisms has the appropriate type because of the naturality square for $\delta(c)$: the domain of $C(\delta_{\sigma_2})(M(c))$ is $C(\theta'(c))(C(\delta_{\sigma_1})(e(\sigma_1)))$, which equals $C(\theta'(\delta_{\sigma_2}))(C(\theta(c))(e(\sigma_1)))$ by naturality of δ and functoriality of C.

The *0-subst assoc/unit* rules are just associativity and unit of functor composition. The *0-resp functoriality* rules are consequences of the functoriality of the type C. *1-subst* and *1-resp* can be calculated from the definition of map. The *def.* map *for* $A[\theta]$ is associativity of whiskering.

Terms It is simple to check that a term $\mathsf{Tm}\ \Delta\ A$ and a functor $\Gamma \longrightarrow \Delta$ can be composed as indicated by $M[\theta]$:

$$(M[\theta])(\sigma) = M(\theta(\sigma))$$
$$(M[\theta])(c) = M(\theta(c))$$

This composition is functorial, as *1-subst assoc/unit* states, because θ is.

$M[\delta]$ is, component-wise, the action on morphisms of the term M, which is required to be functorial by the definition of $\mathsf{Tm}\ \Gamma\ A$, justifying *1-resp assoc/unit* and *1-resp preserves refl*. *1-resp for 1-subst* is an associativity property.

Term Transformations Semantic identity and vertical and horizontal composition for term transformations are defined as follows:

$$(\mathsf{id}_M)_\sigma = \mathsf{id}_{M(\sigma)}$$
$$(\alpha_2 \circ \alpha_1)_\sigma = \alpha_{2\sigma} \circ \alpha_{1\sigma}$$
$$(\alpha[\delta])_\sigma = N(\delta_\sigma) \circ A(c)(\alpha(\theta(\sigma)))$$

Identity $\mathsf{id} : M \Longrightarrow M$ is the pointwise identity; naturality holds by the unit laws. Vertical composition of $\alpha_2 : M_2 \Longrightarrow M_3$ and $\alpha_1 : M_1 \Longrightarrow M_2$ is given pointwise as well; naturality holds using naturality for the components and functoriality. For horizontal composition of $\alpha : M \Longrightarrow N$ and $\delta : \theta \Longrightarrow \theta'$, naturality again holds by naturality of the components and functoriality. Because of these componentwise definitions, it is simple to check that these operations are associative and unital, and that they satisfy an "adjusted" version of the interchange law.

Equality Rules The equality rules in Figure 7.3 are all obvious in the semantics, because equality is real semantic equality, which is a congruence, and everything respects it.

7.3.2 Semantic Contexts

Empty Context The empty context is interpreted as the category 1, which has one object and its identity morphism. Because 1 is discrete (no non-identity morphisms), it is self dual. Because the set of objects and the set of arrows are unital, there is only one functor into 1 (it is a terminal object in Cat), and only one natural transformation between this functor and itself. This justifies the remaining equations.

Covariant term variables $\Gamma\ ,\ x{:}A^+$ is interpreted by the Grothendieck construction $\int_\Gamma A$. As Hofmann and Streicher (1998) discuss, there is a bijection between functors $\theta : \Gamma \longrightarrow \int_\Delta A$ and pairs (θ_1, M) where $\theta_1 : \Gamma \longrightarrow \Delta$ and $M : \mathsf{Tm}\ \Gamma\ A[\theta_1]$, given by pairing and projection at the meta-level: the above functor p projects the first component, $\mathsf{v} : \mathsf{Tm}\ (\int_\Gamma A)\ (A[\mathsf{p}])$ projects the second, and we write (θ_1, M) for the reverse direction. Variables are interpreted as $\mathsf{v}[\mathsf{p}^{\pm n}]$, where $\mathsf{p}^{\pm n}$ is the appropriate number of projections to find the variable in Γ. The transformation rule says that the natural transformations between two functors into $\int_\Gamma A$ are themselves given as pairs, which follows from the fact that morphisms in $\int_\Gamma A$ are pairs: the transformation rule is exactly the definition of morphisms written out type-theoretically. More concretely, we can define a natural transformation of the appropriate type by $(\delta, \alpha)_\sigma = (\delta_\sigma, \alpha_\sigma)$, and that the commutativity condition follows from commutativity for δ and α.

The $\beta\eta$ equations hold because objects and morphisms in $\int_\Gamma A$ are products. *1-id* follows from η. *refl* and *trans* follow from the definition of identity and composition for this category. *1-subst* and *1-resp* and *2-resp* all reassociate compositions. The $^{\mathsf{op}}$ rules requires the definition of contravariant context extension, which we treat next.

Contravariant term variables $\Gamma\ ,\ x{:}A^{-}$ is interpreted as $(\int_{\Gamma^{\mathsf{op}}} A)^{\mathsf{op}}$: We are given a category Γ and a functor $[\![A]\!] : [\![\Gamma]\!]^{\mathsf{op}} \longrightarrow Cat$. Thus, we can form $\int_{[\![\Gamma]\!]^{\mathsf{op}}} [\![A]\!]$. In the syntax, we have weakening for a contravariant context extension, so we require a functor $[\![\Gamma\ ,\ x{:}A^{-}]\!] \longrightarrow [\![\Gamma]\!]$. However, $\mathsf{p} : \int_{[\![\Gamma]\!]^{\mathsf{op}}} [\![A]\!] \longrightarrow [\![\Gamma]\!]^{\mathsf{op}}$ faces in the wrong direction. Thus, we interpret $\Gamma\ ,\ x{:}A^{-}$ as $(\int_{[\![\Gamma]\!]^{\mathsf{op}}} [\![A]\!])^{\mathsf{op}}$. For any functor $F : C \longrightarrow D$, there is a functor $F^{\mathsf{op}} : C^{\mathsf{op}} \longrightarrow D^{\mathsf{op}}$ given by the same data, so $\mathsf{p}^{\mathsf{op}} : (\int_{[\![\Gamma]\!]^{\mathsf{op}}} [\![A]\!])^{\mathsf{op}} \longrightarrow [\![\Gamma]\!]$ provides the required weakening map.

The substitution and transformation rules express the fact that $(\int_{\Gamma} A)^{\mathsf{op}}$ has similar structure to $\int_{\Gamma} A$: objects and morphisms are pairs of a certain form. Indeed, we could directly spell out the construction of a *contravariant fibration* $\int^{-}_{\Gamma} A$ from a contravariant functor $A : \Gamma^{\mathsf{op}} \longrightarrow Cat$, but here we derive the definition from $^{\mathsf{op}}$ instead.[2]

The equations are analogous to those for $\int_{\Gamma} A$. The *invol* rules are clearly validated by this definition using the fact that $\Gamma^{\mathsf{op}\mathsf{op}} = \Gamma$.

7.3.3 Semantic Types

Π Π-types are defined as in Hofmann and Streicher (1998): we follow their construction, checking that everywhere they depend on symmetry of equality, we have inserted the appropriate $^{\mathsf{op}}$'s.

For a category Γ and a $A : \mathsf{Ty}\ \Gamma^{\mathsf{op}}$, we abbreviate semantic contravariant context extension $(\int_{\Gamma^{\mathsf{op}}} A)^{\mathsf{op}}$ by $\Gamma.A^{-}$. Given a $B : \mathsf{Ty}\ \Gamma.A^{-}$ and an object $\sigma \in \mathsf{Ob}\ \Gamma$, we define $B_{\sigma} : \mathsf{Ty}\ A(\sigma)$ by

$$(B_{\sigma})(\sigma') = B(\sigma, \sigma')$$
$$(B_{\sigma})(c) = B(\mathsf{id}_{\sigma}, c)$$

For any Γ and A, the $\mathsf{Tm}\ \Gamma\ A$ are the objects of a category with morphisms given by term transformations α. This lets us define a Π type as follows:

$$(\Pi\ A\ B)_{\sigma} = \mathsf{Tm}\ A(\sigma)^{\mathsf{op}}\ B_{\sigma}$$

Functoriality is given by pre- and post-composition: the contravariance of A ensures that the pre-composition faces the right direction. λ and application and $\beta\eta$ rules are interpreted by giving a bijection between $\mathsf{Tm}\ \Gamma.A^{-}\ B$ and $\mathsf{Tm}\ \Gamma\ \Pi AB$. The transformation intro and elim and $\beta\eta$ rules express a bijection between $M \Longrightarrow N : \Gamma \longrightarrow \Pi AB$ and $M\ \mathsf{v} \Longrightarrow N\ \mathsf{v} : \Gamma.A^{-} \longrightarrow B$. The proof follows Hofmann and Streicher (1998), Section 5.3, which observes that the groupoid interpretation justifies functional extensionality. Simple calculations validate the remaining rules.

Σ Because both subcomponents of $\Sigma\, x{:}A.\ B$ are covariant, the interpretation given in Hofmann and Streicher (1998) adapts to our setting unchanged. In particular, given $A : \mathsf{Ty}\ \Gamma$ and $B : \mathsf{Ty}\ (\int_{\Gamma} A)$, a semantic Σ-type, $\Sigma_A B$ is defined by

$$(\Sigma AB)_o = \int_{A(o)} B_o$$

[2] In fact, it is more common to define $\int^{-}_{\Gamma} A$ so that it is equivalent to $(\int_{\Gamma^{\mathsf{op}}} (\mathsf{op} \circ A))^{\mathsf{op}}$. We have avoided this definition because we have only considered syntactic rules for the op functor on contexts, not on *types*. We will return to this point below.

with functoriality defined componentwise. Pairing and projection, and pairing and projection for transformations, follow from the definition of the Grothendieck construction.

Generic rules for set **and elements** set is interpreted as the constant functor returning $Sets$, the category of sets and functions. We avoid size issues by taking Cat to be the (larger) category of large categories, so that $Sets$ is an object.

Because the action on morphisms of a constant functor is the identity, $\mathsf{Tm}\ \Gamma\ \mathsf{set}$ is bijective with $\Gamma \longrightarrow Sets$. There is a full embedding $discrete : Sets \longrightarrow Cat$. This embedding takes each set to the discrete category on that set: a set X is mapped to the category whose set of objects is X and whose arrows are only the identity on each element of X. This embedding is full because the functors between two discrete categories are bijective with the set-theoretic functions between their objects (the action on morphisms is trivial, because the only maps are identities). Thus, we can represent $\mathcal{D}(S)$ semantically by $discrete \circ S$. As usual, we overload notation and write $\mathcal{D}(S)$ for $discrete \circ S$.

The transformation rule for set expresses (half of) an isomorphism between, on the one hand, natural transformations between two functors into $Sets$, and, on the other, terms $\mathsf{Tm}\ \int_\Gamma \mathcal{D}(S)\ \mathcal{D}(S')$. This direction is implemented as follows: we construct $\widetilde{M}\colon\ S \Longrightarrow S' : \Gamma \longrightarrow Sets$ given $M : \mathsf{Tm}\ (\int_\Gamma(\mathcal{D}(S)))\ (\mathcal{D}(S'))$ by

$$(\widetilde{M})(\sigma) = x \mapsto M(\sigma, x)$$

For a morphism c, naturality is given by $x \mapsto M(c, \mathsf{id}_x)$. This "currying" is exactly analogous to the interpretation of Π types, except here we abstract over a covariant variable.

The transformation introduction rule for $\mathcal{D}(S)$ is just the identity transformation, and the transformation elimination rule is equality reflection. To interpret equality reflection, observe that (1) by proof-irrelevance for equality in the semantics, two $\mathsf{Tm}\ \Gamma\ \mathcal{D}(S)$ are determined entirely by their action on objects, as the action on morphisms produces a proof of equality and (2) the interpretation of the premise says that $M(\sigma) = N(\sigma)$ for all σ. Thus M and N are equal $\mathsf{Tm}\ \Gamma\ \mathcal{D}(S)$.

The *def.* $\mathcal{D}(-)$ equation similarly expresses that the action on morphisms of a $\mathsf{Tm}\ \Gamma\ \mathcal{D}(S)$ proves an equality. The η rule for $Sets$ is analogous to the η rule for Π. The η for for $\mathcal{D}(-)$ holds because all morphisms are the identity. *0-subst* is given compositionally. *0-resp* is the identity for a constant functor, and for $\mathcal{D}(S)$ determined by the action on morphisms of S. The remaining equations are true by simple calculations.

Particular Sets Semantic Π and Σ are defined as follows:

$$\begin{array}{lcl}(\Sigma\ S\ S')(\sigma) & = & \Sigma_{m \in S(\sigma)} S'(\sigma, m) \\ (\Sigma\ S\ S')(c) & = & (m, m') \mapsto (S(c)(m), S(c, \mathsf{id})(m'))\end{array}$$

$$\begin{array}{lcl}(\Pi\ S\ S')(\sigma) & = & \Pi_{m \in S(\sigma)} S'(\sigma, m) \\ (\Pi\ S\ S')(c) & = & f \mapsto (x \mapsto S(c, \mathsf{id})(f(S(c)(x))))\end{array}$$

$\mathsf{0}$ is the empty set, $\mathsf{1}$ is the one-element set, and $\mathsf{2}$ is booleans. For identity, we define $\mathsf{Id}_S\ M\ N : \mathsf{Ty}\ \Gamma$ by

$$\begin{array}{rcll}(\mathsf{Id}_A\ M\ N)(o) & = & 1 & \text{if } M(o) = N(o) \\ & = & 0 & \text{otherwise}\end{array}$$

Functoriality holds because the functorial action of M and N give equations, which can be combined using symmetry and transitivity to give the result.

The term rules for Π and Σ express a bijection between the set-theoretic Π and Σ and the category-theoretic definition applied to discrete categories. The intro maps for $\mathsf{1}$ and $\mathsf{2}$ are the corresponding elements, and the elimination maps are standard for an initial object and a coproduct. Because if and abort eliminate towards an arbitrary type, we check them in more detail. For the covariant if (where $M : \mathsf{Tm}\ \Gamma\ \mathsf{2}$), we proceed as follows:

$$\begin{array}{rcl}(\mathsf{if}(M, M_1, M_2))(\sigma) & = & M_1(\sigma) \text{ if } M(\sigma) \\ (\mathsf{if}(M, M_1, M_2))(\sigma) & = & M_2(\sigma) \text{ otherwise}\end{array}$$

$$\begin{array}{rcl}(\mathsf{if}(M, M_1, M_2))(c : \sigma_1 \longrightarrow \sigma_2) & = & M_1(c) \text{ if } M(\sigma_1) \\ (\mathsf{if}(M, M_1, M_2))(c : \sigma_1 \longrightarrow \sigma_2) & = & M_2(c) \text{ otherwise}\end{array}$$

The action on morphisms is well-typed because $M(c)$ shows that $M(\sigma_1) = M(\sigma_2)$. The contravariant if is defined similarly, as $\mathsf{2}$ is discrete and therefore symmetric. The semantic version of abort is defined by elimination on the empty set. The rules for identity express that the semantics of Id_S is isomorphic to the transformations at $\mathcal{D}(S)$, which are both just the equations between members of $\mathcal{D}(S)$. The remaining equations are validated by calculation.

7.3.4 Soundness Theorem

Next, we formally interpret the syntax of the theory into Cat. We have already discussed the semantic constructions that make up the inductive step of each case, so what remains is to tie it together. We follow the development in Hofmann (1995): first, we define a partial interpretation of raw syntax. This consists of the following judgements, where we write $\overline{\Gamma}$, $\overline{M}$, etc. for semantic terms.

$$\begin{array}{l}\Gamma \gg \overline{\Gamma} \\ \Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta} : \overline{\Delta} \\ \Gamma \gg \overline{\Gamma} \vdash \delta \gg \overline{\delta} : \overline{\theta} \Longrightarrow_{\overline{\Delta}} \overline{\theta'} \\ \Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}\ \mathsf{type} \\ \Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \overline{A} \\ \Gamma \gg \overline{\Gamma} \vdash \alpha \gg \overline{\alpha} : \overline{M} \Longrightarrow_{\overline{A}} \overline{M'}\end{array}$$

The first judgement means that the meaning of Γ is the category $\overline{\Gamma}$. The second means that if the meaning of Γ is $\overline{\Gamma}$, then the meaning of θ is a functor $\overline{\theta}$ into the category $\overline{\Delta}$. The remaining judgements are analogous.

For contexts and types, the rules simply track the structure of the typing derivation, and apply the corresponding semantic construction to the results. For example:

$$\frac{}{\cdot \gg 1} \qquad \frac{\Gamma \gg \overline{\Gamma} \quad \Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}\,\mathsf{type}}{\Gamma\,,\, x{:}A^{+} \gg \textstyle\int_{\overline{\Gamma}} \overline{A}} \qquad \frac{\Gamma \gg \overline{\Gamma} \quad \Gamma^{\mathsf{op}} \gg \overline{\Gamma}^{\mathsf{op}} \vdash A \gg \overline{A}\,\mathsf{type}}{\Gamma\,,\, x{:}A^{-} \gg (\textstyle\int_{\overline{\Gamma}^{\mathsf{op}}} \overline{A})^{\mathsf{op}}} \qquad \frac{\Gamma \gg \overline{\Gamma}}{\Gamma^{\mathsf{op}} \gg \overline{\Gamma}^{\mathsf{op}}}$$

$$\frac{\Gamma^{\mathsf{op}} \gg \overline{\Gamma}^{\mathsf{op}} \vdash A \gg \overline{A}\,\mathsf{type} \quad (\Gamma\,,\, x{:}A^{-}) \gg (\textstyle\int_{\overline{\Gamma}^{\mathsf{op}}} \overline{A})^{\mathsf{op}} \vdash B \gg \overline{B}\,\mathsf{type}}{\Gamma \gg \overline{\Gamma} \vdash \Pi\, x{:}A.\, B \gg \Pi_{\overline{A}} \overline{B}\,\mathsf{type}}$$

For the remaining judgements, the rules do type checking in the semantics, in that they insist that the semantic types fit together appropriately. For example, we show the rules for Π-types:

$$\frac{\begin{array}{l}\Gamma^{\mathsf{op}} \gg \overline{\Gamma}^{\mathsf{op}} \vdash A \gg \overline{A}\,\mathsf{type} \\ (\Gamma\,,\, x{:}A^{-}) \gg (\textstyle\int_{\overline{\Gamma}^{\mathsf{op}}} \overline{A})^{\mathsf{op}} \vdash B \gg \overline{B}\,\mathsf{type} \\ (\Gamma\,,\, x{:}A^{-}) \gg (\textstyle\int_{\overline{\Gamma}^{\mathsf{op}}} \overline{A})^{\mathsf{op}} \vdash M \gg \overline{M} : \overline{B}\end{array}}{\Gamma \gg \overline{\Gamma} \vdash \lambda\, x{:}A.\, M^{B} \gg \lambda(\overline{M}) : \Pi_{\overline{A}} \overline{B}} \qquad \frac{\begin{array}{l}\Gamma^{\mathsf{op}} \gg \overline{\Gamma}^{\mathsf{op}} \vdash A \gg \overline{A}\,\mathsf{type} \\ (\Gamma\,,\, x{:}A^{-}) \gg (\textstyle\int_{\overline{\Gamma}^{\mathsf{op}}} \overline{A})^{\mathsf{op}} \vdash B \gg \overline{B}\,\mathsf{type} \\ \Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \Pi_{\overline{A}} \overline{B} \\ \Gamma^{\mathsf{op}} \gg \overline{\Gamma}^{\mathsf{op}} \vdash N \gg \overline{N} : \overline{A}\end{array}}{\Gamma \gg \overline{\Gamma} \vdash \mathsf{app}_{A,x.B}(M, N) \gg \overline{M}\ \overline{N} : \overline{B}[\mathsf{id}_{\Gamma}, \overline{N}]}$$

The transformation rules are similar.

A simple induction shows that the interpretation is unique if it exists:

LEMMA 7.3.3: UNIQUENESS.

- *If* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \gg \overline{\Gamma_1}$ *then* $\overline{\Gamma} = \overline{\Gamma}_1$.
- *If* $\Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta} : \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta_1} : \overline{\Delta_1}$ *then* $\overline{\Delta} = \overline{\Delta_1}$ *and* $\overline{\theta} = \overline{\theta_1}$.
- *If* $\Gamma \gg \overline{\Gamma} \vdash \delta \gg \overline{\delta} : \overline{\theta} \Longrightarrow_{\overline{\Delta}} \overline{\theta'}$
  and $\Gamma \gg \overline{\Gamma} \vdash \delta \gg \overline{\delta_1} : \overline{\theta_1} \Longrightarrow_{\overline{\Delta_1}} \overline{\theta'_1}$
  then $\Delta = \overline{\Delta_1}$ *and* $\theta = \overline{\theta_1}$ *and* $\overline{\theta'} = \overline{\theta'_1}$ *and* $\overline{\delta} = \overline{\delta_1}$.
- *If* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}\,\mathsf{type}$ *and* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A_1}\,\mathsf{type}$ *then* $\overline{A} = \overline{A_1}$.
- *If* $\Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \overline{A}$ *and* $\Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M_1} : \overline{A_1}$ *then* $\overline{A} = \overline{A_1}$ *and* $\overline{M} = \overline{M_1}$.
- *If* $\Gamma \gg \overline{\Gamma} \vdash \alpha \gg \overline{\alpha} : \overline{M} \Longrightarrow_{\overline{\Delta}} \overline{M'}$
  and $\Gamma \gg \overline{\Gamma} \vdash \alpha \gg \overline{\alpha_1} : \overline{M_1} \Longrightarrow_{\overline{A_1}} \overline{M'_1}$
  then $\overline{A} = \overline{A_1}$ *and* $\overline{M} = \overline{M_1}$ *and* $\overline{M'} = \overline{M'_1}$ *and* $\overline{\alpha} = \overline{\alpha_1}$.

The proof exploits the fact that the type of a term is determined (up to equality) by its immediate subterms. Otherwise, there would be some non-determinism in e.g. the application rule, which would require guessing a domain type, and this non-determinism would make the interpretation less obviously unique.

The interpretation interacts with weakening in the expected way: First, a weakening $w : \Gamma \supseteq \Delta$ induces a functor between the interpretations of the contexts. Second, weakening commutes with the interpretation, in the sense that interpreting in a larger context is the same as interpreting in a smaller context and then weakening.

LEMMA 7.3.4: WEAKENING.

1. *If* $w : \Gamma \supseteq \Delta$ *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Delta \gg \overline{\Delta}$ *then there is a functor* $\overline{w} : \Gamma \longrightarrow \Delta$
2. *For any of the contextual interpretations, if* $\Gamma\,\mathsf{ctx}$ *and* $\Delta\,\mathsf{ctx}$ *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Delta \gg \overline{\Delta}$ *and* $w : \Gamma \supseteq \Delta$ *and* $\Delta \gg \overline{\Delta} \vdash t \gg \overline{t} : \overline{J}$ *then* $\Gamma \gg \overline{\Gamma} \vdash t \gg \overline{t}[\overline{w}] : \overline{J}[\overline{w}]$

Proof. The construction for the first part was described above. The second part is true by a simple induction, using the equations discussed above for how semantic substitution commutes with constructors. □

THEOREM 7.3.5: SOUNDNESS.

Formation:

- *If* Γ ctx *then* $\Gamma \gg \overline{\Gamma}$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash \theta : \Delta$ *then* $\Delta \gg \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta} : \overline{\Delta}$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash \delta : \theta \Longrightarrow_{\Delta} \theta'$ *then* $\Delta \gg \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta} : \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta' \gg \overline{\theta'} : \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \delta \gg \overline{\delta} : \overline{\theta} \Longrightarrow_{\overline{\Delta}} \overline{\theta'}$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash A$ type *then* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}$ type
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash M : A$ *then* $\Gamma \gg \overline{\Gamma} \vdash A \gg A$ type *and* $\Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \overline{A}$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash \alpha : M \Longrightarrow_A M'$ *then* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}$ type *and* $\Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \overline{A}$ *and* $\Gamma \gg \overline{\Gamma} \vdash M' \gg \overline{M'} : \overline{A}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \alpha \gg \overline{\alpha} : \overline{M} \Longrightarrow_{\overline{A}} \overline{M'}$

Equality:

- *If* $\Gamma \equiv \Delta$ *then* $\Gamma \gg \overline{\Gamma}$ *and* $\Delta \gg \overline{\Delta}$ *and* $\overline{\Gamma} = \overline{\Delta}$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash \theta \equiv \theta' : \Delta$ *then* $\Delta \gg \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta} : \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta' \gg \overline{\theta}' : \overline{\Delta}$ *and* $\overline{\theta} = \overline{\theta}'$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash \delta \equiv \delta' : \theta \Longrightarrow_{\Delta} \theta'$ *then* $\Delta \gg \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta \gg \overline{\theta} : \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \theta' \gg \overline{\theta}' : \overline{\Delta}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \delta \gg \overline{\delta} : \overline{\theta} \Longrightarrow_{\overline{\Delta}} \overline{\theta}'$ *and* $\Gamma \gg \overline{\Gamma} \vdash \delta' \gg \overline{\delta}' : \overline{\theta} \Longrightarrow_{\overline{\Delta}} \overline{\theta}'$ *and* $\overline{\delta} = \overline{\delta}'$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash A \equiv A'$ type *then* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}$ type *and* $\Gamma \gg \overline{\Gamma} \vdash A' \gg \overline{A}'$ type *and* $\overline{A} = \overline{A}'$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash M \equiv M' : A$ *then* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}$ type *and* $\Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \overline{A}$ *and* $\Gamma \gg \overline{\Gamma} \vdash M' \gg \overline{M}' : \overline{A}$ *and* $\overline{M} = \overline{M}'$
- *If* Γ ctx *and* $\Gamma \gg \overline{\Gamma}$ *and* $\Gamma \vdash \alpha \equiv \alpha' : M \Longrightarrow_A M'$ *then* $\Gamma \gg \overline{\Gamma} \vdash A \gg \overline{A}$ type *and* $\Gamma \gg \overline{\Gamma} \vdash M \gg \overline{M} : \overline{A}$ *and* $\Gamma \gg \overline{\Gamma} \vdash M' \gg \overline{M}' : \overline{A}$ *and* $\Gamma \gg \overline{\Gamma} \vdash \alpha \gg \overline{\alpha} : \overline{M} \Longrightarrow_{\overline{A}} \overline{M}'$ *and* $\Gamma \gg \overline{\Gamma} \vdash \alpha' \gg \overline{\alpha}' : \overline{M} \Longrightarrow_{\overline{A}} \overline{M}'$ *and* $\overline{\alpha} = \overline{\alpha}'$

Proof. Mutual induction on the principal formation/equality judgement. □

There is no need to prove a separate compositionality lemma (substitution commutes with interpretation), because this is expressed by the interpretation of explicit substitutions. The usual proof of compositionality is embedded in the proof that each of the *0-subst* and *1-subst* equations hold.

On the semantic front, an important piece of future work is to generalize this semantics to a class of 2-categories with appropriate structure (for example, the role of functors into Cat will be played by fibrations), prove a completeness result, and investigate other instances of this structure.

7.4 Discussion

In this chapter, we have taken the first steps towards a directed dependent type theory, where each type is equipped with a notion of transformation between its members, and dependent types and terms are equipped with a functorial action on transformations. This calculus has several interesting technical ingredients: First, we represent transformations as a judgement, rather than as an analogue of the identity type, which is necessary because the collection of transformations is not functorial in the ambient context. Second, we track the variances of assumptions, which is necessary to ensure that the functorial action of each type constructor can be defined. Third, we internalize the identity and composition principles of a 2-category as identity and substitution operations, whose meaning is explained by definitional equality rules. These principles include familiar explicit substitutions in terms and types, as well as the functorial action of each type constructor (map), and operations stating that terms and transformations respect transformation (e.g. $M[\delta]$ and $\alpha[\delta]$). The definitional equality rules explain these principles in terms of other constructs of the calculus. An interesting avenue for future work is to give a more syntactic presentation, where the identity and composition principles are treated as meta-operations (like ordinary substitution traditionally is). We conjecture that the equations given here are sufficient to define these meta-operations, but we leave an investigation of this approach to future work.

In the next chapter, we show how 2DTT accounts for the structural properties of the generic judgement, and we sketch other extensions that will make it useful for programming with logical systems that mix admissibility and derivability.

Chapter 8

Applications and Extensions

In this chapter, we discuss some applications of 2DTT to programming with variable binding. First, we demonstrate that 2DTT provides the right vocabulary for analyzing the structural properties of the generic judgement. Second, we sketch a variety of extensions to the base theory that will make 2DTT a convenient language for programming with logics.

8.1 The Generic Judgement

In this section, we demonstrate two facts: First, 2DTT provides the tools to describe the structural properties of the generic judgement. Second, 2DTT provides general answers to the question of what operations can be used in the subject of a generic judgement, while remaining structural.

8.1.1 Simple Contexts and Variables

Here, we concentrate on the structural properties of weakening, exchange, and contraction; substitution is discussed below. To illustrate these ideas in as simple a setting as possible, we begin by representing uni-sorted contexts (where the lone sort is written i for individual), which are essentially natural numbers representing the number of free variables of an expression. Contexts are a directed type, with morphisms given by variable-for-variable substitutions. The generalization to multi-sorted contexts is analogous.

Contexts are constructed by ϵ (zero) and Ψ, i (successor). A transformation $\Psi \Longrightarrow_{\mathsf{ctx}} \Psi'$ means that $\Psi \vdash \Psi'$, so syntax will typically be indexed contravariantly by Ψ. Transformations between contexts have two introduction forms, ϵ and (α, v), and one elimination form (first projection). map at $\mathsf{var}(-)$ plays the role of second projection. Variables are classified by the set $\mathsf{var}(\Psi)$, and introduced by zero and successor indices.

These constructs satisfy the following equations: For contexts, the β-rule says that projecting from a pair is the first component, and the η-rules expand renamings into concrete contexts. *1-subst* is standard. The action of the constructors on renamings, given by *1-resp*, is trivial for ϵ and uses "parallel" renaming extension

$$((- \circ \pi), \mathsf{z}) : \Psi \Longrightarrow \Psi' \longrightarrow \Psi, \mathsf{i} \Longrightarrow \Psi', \mathsf{i}$$

for Ψ, i. We discuss the remaining equations below.

$$\frac{}{\Gamma \vdash \mathsf{ctx}\ \mathsf{type}} \quad \frac{}{\Gamma \vdash \epsilon : \mathsf{ctx}} \quad \frac{\Gamma \vdash \Psi : \mathsf{ctx}}{\Gamma \vdash \Psi, \mathsf{i} : \mathsf{ctx}}$$

$$\frac{}{\Gamma \vdash \epsilon : \Psi \Longrightarrow_{\mathsf{ctx}} \epsilon} \quad \frac{\Gamma \vdash \alpha : \Psi \Longrightarrow_{\mathsf{ctx}} \Psi' \quad \Gamma \vdash v : \mathcal{D}(\mathsf{var}(\Psi))}{\Gamma \vdash (\alpha, v) : \Psi \Longrightarrow_{\mathsf{ctx}} \Psi', \mathsf{i}} \quad \frac{}{\Gamma \vdash \pi : \Psi, \mathsf{i} \Longrightarrow_{\mathsf{ctx}} \Psi}$$

$$\frac{\Gamma^{\mathsf{op}} \vdash \Psi : \mathsf{ctx}}{\Gamma \vdash \mathsf{var}(\Psi) : \mathsf{set}} \quad \frac{}{\Gamma \vdash \mathsf{z} : \mathcal{D}(\mathsf{var}(\Psi, \mathsf{i}))} \quad \frac{\Gamma \vdash v : \mathcal{D}(\mathsf{var}(\Psi))}{\Gamma \vdash \mathsf{s}(v) : \mathcal{D}(\mathsf{var}(\Psi, \mathsf{i}))}$$

Equations for contexts:

$$\begin{array}{rcll}
\pi \circ (\alpha, v) & \equiv & \alpha & \textit{2-}\beta \\
\alpha : \Psi \Longrightarrow \epsilon & \equiv & \epsilon & \textit{2-}\eta \\
\alpha : \Psi \Longrightarrow \Psi', \mathsf{i} & \equiv & (\pi \circ \alpha), \mathsf{z} &
\end{array}$$

$$\begin{array}{rcll}
\mathsf{ctx}[\theta] & \equiv & \mathsf{ctx} & \textit{0-subst} \\
\mathsf{map}_{\Delta.\mathsf{ctx}}\ \delta\ M & \equiv & M & \textit{0-resp}
\end{array}$$

$$\begin{array}{rcll}
\epsilon[\theta] & \equiv & \epsilon & \textit{1-subst} \\
(\Psi, \mathsf{i})[\theta] & \equiv & \Psi[\theta], \mathsf{i} & \\
\epsilon[\delta] & \equiv & \epsilon & \textit{1-resp} \\
(\Psi, \mathsf{i})[\delta] & \equiv & (\Psi[\delta] \circ \pi), \mathsf{z} &
\end{array}$$

$$\begin{array}{rcll}
\mathsf{refl}_{\epsilon} & \equiv & \epsilon & \textit{refl} \\
\mathsf{refl}_{\Psi,\mathsf{i}} & \equiv & (\mathsf{refl}_{\Psi}) \circ \pi, \mathsf{z} & \textit{refl} \\
\epsilon \circ \alpha & \equiv & \epsilon & \textit{trans} \\
(\alpha_2, v) \circ \alpha_1 & \equiv & \alpha_2 \circ \alpha_1, \mathsf{map}\ \alpha_1\ v &
\end{array}$$

$$\begin{array}{rcll}
\epsilon[\delta] & \equiv & \epsilon & \textit{2-resp} \\
(\alpha, v)[\delta] & \equiv & \alpha[\delta], v[\theta_1] & \\
\pi[\delta] & \equiv & \pi &
\end{array}$$

Equations for variables:

$$\begin{array}{rcll}
\mathsf{var}(\Psi)[\theta] & \equiv & \mathsf{var}(\Psi[\theta]) & \textit{1-subst} \\
\mathsf{var}(\Psi)[\delta] & \equiv & x.\mathsf{map}_{\psi.\mathcal{D}(\mathsf{var}(\psi))}\ \Psi[\delta^{\mathsf{op}}]\ x & \textit{1-resp} \\
\mathsf{map}_{\psi.\mathcal{D}(\mathsf{var}(\psi))}\ (\alpha, v)\ \mathsf{z} & \equiv & v & \textit{1-resp} \\
\mathsf{map}_{\psi.\mathcal{D}(\mathsf{var}(\psi))}\ (\alpha, v)\ \mathsf{s}(i) & \equiv & \mathsf{map}_{\psi.\mathcal{D}(\mathsf{var}(\psi))}\ \alpha\ \mathsf{i} & \textit{1-resp} \\
\mathsf{map}_{\psi.\mathcal{D}(\mathsf{var}(\psi))}\ \pi\ i & \equiv & \mathsf{s}(i) & \textit{1-resp}
\end{array}$$

$$\begin{array}{rcll}
\mathsf{z}[\theta] & \equiv & \mathsf{z} & \textit{1-subst} \\
\mathsf{s}(v)[\theta] & \equiv & \mathsf{s}(v[\theta]) & \textit{1-subst} \\
\mathsf{z}[\delta] & \equiv & \star & \textit{1-resp} \\
\mathsf{s}(v)[\delta] & \equiv & \star & \textit{1-resp}
\end{array}$$

Figure 8.1: Simple contexts and variables

8.1.2 Structurality

To illustrate structurality, we consider a representation of a judgement $\Psi \vdash \phi$, where Ψ is a ctx and ϕ is a proposition that can mention the variables in Ψ. We represent propositions by a set

$$\frac{\Gamma^{\mathsf{op}} \vdash \Psi : \mathsf{ctx}}{\Gamma \vdash \mathsf{propo}\ \Psi : \mathsf{set}}$$

This typing says that propositions are contravariantly functorial in Ψ, meaning that

$$\frac{w : \Psi \Longrightarrow_{\mathsf{ctx}} \Psi' \quad \phi : \mathsf{propo}\ \Psi'}{(\mathsf{map}_{\psi^-.\mathsf{propo}\ \psi}\ (w^-/\psi)\ e) : \mathsf{propo}\ \Psi}$$

Moreover, the functoriality equations for map stipulate that weakening by the identity is the identity, weakening by a composition is composition of the weakenings, and so on. We will sometimes abbreviate $\mathsf{map}_{\psi^-.\mathsf{propo}\ \psi}\ w^-/\psi\ e$ by $\mathsf{map}\ w\ e$ when the meaning is clear from context.

Next, we represent natural deduction derivations by a type

$$\frac{\Gamma^{\mathsf{op}} \vdash \Psi : \mathsf{ctx} \quad \Gamma \vdash \phi : \mathsf{prop}\ \Psi}{\Gamma \vdash \mathsf{nd}\ \Psi\ \phi : \mathsf{set}}$$

The type-generic rule for map specializes to the appropriate renaming principle:

$$\frac{w : \Psi \Longrightarrow_{\mathsf{ctx}} \Psi' \quad e : \mathsf{nd}\ \Psi'\ \phi}{(\mathsf{map}_{\psi^-, a^+.\mathsf{nd}\ \psi\ a}\ (w^-/\psi, \mathsf{refl}^+/a)\ e) : \mathsf{nd}\ \Psi\ (\mathsf{map}\ w\ \phi)}$$

As desired, this principle says that the renaming of the derivation proves the renaming of the judgement.

8.1.3 Subjects of Judgements

Above, we raised questions about what propositions ϕ may be used in inference rules for $\Psi \vdash \phi$ without invalidating the structural properties. 2DTT provides a general framework for answering these questions, and in fact, the answer is quite simple. As we explain further below, an inference rule will typically be stated for an unknown context variable $\psi : \mathsf{ctx}^-$. Thus, typical instances of propositions in inference rules will be in an unknown context, as in $\psi : \mathsf{ctx}^- \vdash \phi : \mathsf{propo}\ \psi$, or in the extension of an unknown context, as in $\psi : \mathsf{ctx}^- \vdash \phi : \mathsf{propo}\ (\psi, \mathsf{i})$ (e.g. in the premise of the universal quantifier introduction rule). And indeed, *any* terms of these types can be used in rules. The reason is that the term typing judgement in 2DTT provides the necessary guarantees.

This is easiest to explain for a rule

$$\frac{\psi \vdash \phi_1 \ldots \psi \vdash \phi_n}{\psi \vdash c(\phi_1, \ldots, \phi_n)}$$

To prove renaming for a judgement defined by this rule, it is necessary to show that given $w : \Psi_1 \Longrightarrow_{\mathsf{ctx}} \Psi_2$ and propositions $\phi_i : \mathsf{propo}\ \Psi_2$

$$\mathsf{map}\ w\ c_{\Psi_2}(\phi_1, \ldots, \phi_n) \equiv c_{\Psi_1}(\mathsf{map}\ w\ \phi_1, \ldots, \mathsf{map}\ w\ \phi_n)$$

However, *any* term

$$\psi : \mathsf{ctx}^-, \phi_1 : \mathsf{propo}\ \psi^+, \ldots \vdash c : \mathsf{propo}\ \psi$$

has exactly this property, because there is a transformation

$$(w^-/\psi, \mathsf{refl}^+/\phi_1, \ldots) : (\Psi_2{}^-/\psi, \phi_1{}^+/\phi_1, \ldots) \Longrightarrow_{(\psi : \mathsf{ctx}^-, \phi_1 : \mathsf{propo}\ \psi^+, \ldots)} (\Psi_1{}^-/\psi, (\mathsf{map}\ w\ \phi_1)^+/\phi_1, \ldots)$$

and thus functoriality of c gives the result: $c[w, \mathsf{refl}, \ldots]$ gives a transformation at $\mathcal{D}(-)$, which by equality reflection proves the equation.

Moreover, there is nothing special about the indices of the premises being tuples of propositions: any term $\psi : \mathsf{ctx}^- \vdash c : A$ commutes with renaming in a sense appropriate for A.

Context-parametric terms The above discussion has reduced the problem of what propositions can appear in inference rules to deciding what terms of the form $\psi : \mathsf{ctx}^- \vdash M : A$ are permissible in 2DTT. Based on the above examples of programming with logics, we have some intuition for what should and should not be allowed:

- A proposition constructed out of variables and constructors *should* be allowed (e.g. the conclusion of conjunction introduction).
- A proposition constructed by applying the structural properties *should* be allowed (e.g. the conclusion of $\forall$-elimination)
- A proposition constructed by case-analyzing the context ψ *should not* be allowed.

Intuitively, constructors and variables commute with renaming, and renamings commute because they compose. However, a term that intensionally analyzes the context may not commute with renaming, because it may compute differently on the different contexts.

The functoriality component of the definition of terms in 2DTT allows the former two but disallows the latter. Variables, constructors, and map are all ways of constructing a term. However, we cannot, for example, add the usual recursor for ctx, as this would permit functions that do not respect renaming:

$$\begin{aligned}&\Psi : \mathsf{ctx}^+ \vdash \mathsf{tail}(\Psi) : \mathsf{ctx}\\ &\mathsf{tail}(\epsilon) = \epsilon\\ &\mathsf{tail}(\Psi, \mathsf{i}) = \Psi\end{aligned}$$

tail does not respect transformation, in that if $\Psi \Longrightarrow \Psi'$ there is not necessarily a transformation $\mathsf{tail}(\Psi) \Longrightarrow \mathsf{tail}(\Psi')$. To see this, take $\Psi' = (\epsilon, \mathsf{i})$ and $\Psi = (\epsilon, \mathsf{i}, \mathsf{i})$, and let the original renaming map both variables in Ψ to the one variable in Ψ'. There is no renaming $\epsilon \Longrightarrow \mathsf{i}$, so this renaming cannot be preserved by tail. Because the functoriality component of a terms requires it to respect transformation, tail cannot have this type. However, certain functions on contexts do respect transformation. An example is the transformation π, which can be defined by recursion on the context.

Similar considerations influence the elimination forms for variables and syntax: the richer the notion of transformation, the fewer eliminations respect it. For example, we cannot give the standard case-analysis principle for variables $\mathsf{var}(\Psi)$, giving a case for z and a case for s. The reason is that functoriality imposes a coherence constraint between the transformation of

the result of the zero case and the result of the successor case. However, some functions are permissible. For example, we can provide an equality test:

$$\Psi : \mathsf{ctx}^{-}, x, y : \mathsf{var}(\Psi)^{+} \vdash \mathsf{eq}(x, y) : 2'$$

if we take $2'$ not to be the discrete category on booleans, but the two-point category with $\mathsf{false} \leq \mathsf{true}$—equal variables stay equal under renaming, but disequal variables may become equal. Conversely, restricting the transformations at a type enables more general elimination forms: for example, Pouillard and Pottier (2010) show that the operation

$$\mathit{islast} : \mathsf{var}(\Psi, \mathsf{i}) \to 2$$

respects *bijections* on variables, but it does not respect arbitrary renamings.

For syntax, when ctx is restricted to renamings, propo can be equipped with the usual recursor, because case-analysis on constructors commutes with renaming. However, if the context category is instead taken to be term-for-variable substitutions, then this places additional restrictions on a context-polymorphic term: a general recursor is no longer permitted, because a recursive function over propositions does not necessarily respect substitution. On the other hand, arbitrary recursive functions over syntax are classified by the type

$$\Pi\, \psi{:}!\mathsf{ctx}.\ \mathsf{exp}\ \mathsf{un}!\ {}^{-}\psi \to \mathsf{exp}\ \mathsf{un}!\ {}^{+}\psi$$

$!A$, which is discussed in more detail below, *discretizes* a type, forgetting all transformation structure. When A is closed, like ctx, it has both co- and contravariant inclusions into A, which we notate $\mathsf{un}!^{+}$ and $\mathsf{un}!^{-}$. Because transformations at $!\mathsf{ctx}$ are only equalities, a function of this type may avail itself of the usual recursion operators to analyze contexts and expressions.

2DTT provides a natural setting to explore this hierarchy of coarser and finer notions of a transformation on a type. In the semantics, a term can be defined giving a function on raw objects, in addition to a proof that it respects transformation. We can expose this in the syntax in a manner similar to quotient types: one can define a function on a higher-dimensional type by defining a function on the raw terms and proving separately that it respects transformation. We speculate on this extension below.

At present, because we have not yet exposed this functionality in the syntax, ctx and $\mathsf{hctx}(\Psi)$ deviate from the methodology we espoused above: For example, we include π as a primitive elimination form for context transformations—in a more general setting, it could be defined by recursion on contexts and proved to respect transformation. refl, $\alpha \circ \alpha'$, and map are not in general definable in terms of other constructs—though they are for canonical transformations. For variables, the only unusual rules are for *1-resp* and map, which are anomalous because we do not have a general elimination form for variables. Instead, we cheat by (1) reducing $\mathsf{var}(\Psi)[\delta]$ to map and (2) giving rules for map that pattern-match on the term it is applied to.

8.2 Extensions

In this section, we sketch various extensions to 2DTT that will make it more convenient for programming with logics.

8.2.1 Datatypes

Our goal in this section is to show that if we extend 2DTT with a datatype mechanism, then we can generate weakening, exchange, and contraction for many interesting signatures *just by writing down the type in 2DTT*.

For this section, we assume a datatype mechanism that allows definitions of the form

$$\mathsf{D} : (A \to \mathsf{set}) \cong B$$

where A is a type and $\mathsf{D} : (A \to \mathsf{set})^{+} \vdash B : A \to \mathsf{set}$. The idea is that D is an inductive set indexed by A, with unrolling given by B. Inductive types are functorial if their unrollings are, and map at $\mathcal{D}(\mathsf{D}\ a)$ is given recursively using functoriality of its unrolling $\mathcal{D}(B\ a)$. Datatypes of this form should admit the usual induction principles as elimination forms.

A definition of this form could be represented by an indexed μ:

$$\frac{\Gamma^{\mathsf{op}} \vdash A\ \mathsf{type} \quad \Gamma^{\mathsf{op}} \vdash I : A \quad \Gamma, r : \mathcal{D}(A \to \mathsf{set})^{+}, i : A^{-} \vdash F : \mathsf{set} \quad \Gamma^{\mathsf{op}} \vdash M : A}{\Gamma \vdash \mu_A(r.i.F, I) : \mathsf{set}}$$

which is introduced by unrolling

$$\frac{\Gamma \vdash M : F[\mathsf{id}, \lambda\, x.\, \mu(r.i.F, x)^{-}/r, I^{-}/i]}{\Gamma \vdash \mathsf{in}\, M : \mu_A(r.i.F, I)}$$

and eliminated by recursion.

The problem with this definition is that it admits too many recursive types: The variance of r ensures that the recursive occurrences are only in positive positions, which precludes examples such as $D \cong (D \to D)$. However, it still allows some constructions that do not make sense in $Sets$, like $D \cong (D \to 2) \to 2$, which is impossible for size reasons.

One possible solution is to change the intended semantics, say to categories in domains, and allow this formation rule for recursive types. Another is to restrict the recursor to *strictly positive* types. This can be achieved syntactically by inspecting F, checking that r does not occur to the left of an arrow, or semantically by formalizing a type of (indexed) strictly positive functors (Abbott et al., 2005; Chapman et al., 2010; Martin-Löf, 1975), whose fixed points are equivalent to W-types. Because the encoding of indexed W-types as W-types uses propositional equality, it may be useful to consider indexed containers (Altenkirch and Morris, 2009; Gambino and Hyland, 2004) as a primitive notion. Alternatively, we could first explore directed hom types, and then investigate whether the encoding carries over to a higher-dimensional and directed setting.

For the remainder of this section, we will informally check that the definitions are strictly positive.

Simply-typed Signatures

Expressions: Purely Positive Syntax λ-terms are described by the usual de Bruijn datatype:

$$\begin{aligned} &\mathsf{exp} : \mathsf{ctx} \to \mathsf{set} \\ &\mathsf{exp} \ \cong\ \lambda\, \psi.\, (\mathsf{var}(\psi) + (\mathsf{exp}\ \psi \times \mathsf{exp}\ \psi) + \mathsf{exp}\ (\psi, \mathsf{i})) \end{aligned}$$

It is instructive to type-check the right-hand side: our goal is to show under the assumption $\psi : \mathsf{ctx}^-$, the body is a set. By the rules for $+$ (which can be encoded with its usual rules using Σ and booleans) and $\times$ (a degenerate Σ) this is true if $\mathsf{var}(\psi)$and $\mathsf{exp}\ \psi$ and $\mathsf{exp}\ (\psi, \mathsf{i})$ are sets under this assumption. But both var and exp are indexed contravariantly—for var, this is expressed in the formation rule, and for exp, this is expressed by the $\to$ in its type (which is a degenerate Π, and arguments to such a function are a contravariant position). Thus, this reduces to showing that $\psi : \mathsf{ctx}$ and $\psi, \mathsf{i} : \mathsf{ctx}$ in $(\psi : \mathsf{ctx}^-)^{\mathsf{op}}$, which is $(\psi : \mathsf{ctx}^+)$, which permits the variable to be used.

The functorial action of this datatype implements weakening, exchange, and contraction: If $w : \Psi \Longrightarrow \Psi'$ and $e : \mathsf{exp}\ \Psi'$ then $(\mathsf{map}\ w\ e) : \mathsf{exp}\ \Psi$. Moreover, the generic equations of the calculus ensure that weakening by the identity is the identity, weakening by a composition is composition of the weakenings, and so on.

Arithmetic Expressions: Mixed Variance Syntax Similarly, we can define mixed-variance syntax, combining admissibility and derivability, as in the arith example above (assuming a set of natural numbers with its usual rules):

$$\begin{array}{lcl} \mathsf{arith} & : & \mathsf{ctx} \to \mathsf{set} \\ \mathsf{arith} & \cong & \lambda\, \psi.\, \mathsf{var}(\psi) \\ & & +\mathsf{nat} \\ & & +(\mathsf{arith}\ \psi \times (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \times \mathsf{arith}\ \psi) \\ & & +(\mathsf{arith}\ \psi \times \mathsf{arith}\ (\psi, \mathsf{i})) \end{array}$$

The body type-checks because ψ is only used in positions of the appropriate variance—for example, if we put $\mathsf{arith}\ \psi$ to the left an arrow, then the index of arith would again be a negative position, and the variable would not be available for use.

This description of the datatype captures the fact than nat is constant in ψ, rather than relying on subordination-based weakening/strengthening, as above. Below, we will discuss an example where a type is indexed by two contexts of different variances.

Dependently typed syntax

Propositional Logic: Intrinsically Simply-Typed Syntax Next, we consider some examples that go beyond the framework described in Chapter 5. First, we consider typed syntax, in the special case where the types are not dependent on variables. This covers propositional logic and intrinsic encodings of simply typed languages. We assume a set ty of object-language types, e.g. constructed by

$$\mathsf{arr}(\tau_1 : \mathsf{ty}, \tau_2 : \mathsf{ty}) : \mathsf{ty}$$

For this example, we also need to generalize the type ctx to lists of ty's, rather than lists of unit, and generalize variables $\mathsf{var}(\Psi)$ to $\tau \in \Psi$. As with booleans, there is an inductive identity function $-^- : \mathcal{D}(\mathsf{ty} \to \mathsf{ty})$ that allows a contravariant ty assumption to be used contravariantly.

Consider the following Agda datatype:

```
data exp : ctx → ty where
  var : ∀ {Ψ τ} → τ ∈ Ψ → exp Ψ τ
```

$\mathsf{lam} : \forall\ \{\Psi\ \tau 1\ \tau 2\} \to \mathsf{exp}\ (\Psi\ ,\ \tau 1)\ \tau 2 \to \mathsf{exp}\ \Psi\ (\mathsf{arr}\ (\tau 1\ ,\ \tau 2))$
$\mathsf{app} : \forall\ \{\Psi\ \tau 1\ \tau 2\} \to \mathsf{exp}\ \Psi\ (\mathsf{arr}\ (\tau 1\ ,\ \tau 2)) \to \mathsf{exp}\ \Psi\ \tau 1 \to \mathsf{exp}\ \Psi\ \tau 2$

This is a "Catholic" inductive family[1], which means that the indices may vary in the result of each constructor. Catholic families can be made "Protestant" (meaning the indices may vary in recursive calls, but not results) by representing the index constraints as explicit equality proofs:

$\mathsf{data}\ \mathsf{exp}\ (\Psi : \mathsf{ctx})\ (\tau : \mathsf{ty})\ \mathsf{where}$
$\mathsf{var} : \tau \in \Psi \to \mathsf{exp}\ \Psi\ \tau$
$\mathsf{lam} : \forall\ \{\Psi\ \tau 1\ \tau 2\} \to \mathsf{Id}\ \tau\ (\mathsf{arr}\ (\tau 1\ ,\ \tau 2)) \to \mathsf{exp}\ (\Psi\ ,\ \tau 1)\ \tau 2 \to \mathsf{exp}\ \Psi\ \tau$
$\mathsf{app} : \forall\ \{\Psi\ \tau 1\} \to \mathsf{exp}\ \Psi\ (\mathsf{arr}\ (\tau 1\ ,\ \tau)) \to \mathsf{exp}\ \Psi\ \tau 1 \to \mathsf{exp}\ \Psi\ \tau$

Our above notion of datatype allows only Protestant families, which avoids any tacit use of propositional equality—a treatment of Catholic families in the directed and higher-dimensional settings is an interesting subject for future work. However, when the index constraints are on elements of sets, we can encode Catholic families using Id. For example, the Protestant definition of exp can be rendered directly in 2DTT:

$$\begin{array}{lcl} \mathsf{exp} & : & (\mathsf{ctx} \times \mathcal{D}(\mathsf{ty})) \to \mathsf{set} \\ \mathsf{exp} & \cong & \lambda\,(\psi, \tau).\, \tau^{\text{-}} \in \psi \\ & & \quad +\Sigma\, \tau_1, \tau_2{:}\mathcal{D}(\mathsf{ty}).\, \mathsf{Id}\ \tau^{\text{-}}\ \mathsf{arr}(\tau_1, \tau_2) \times \mathsf{exp}\ ((\psi, \tau_1), {\tau_2}^{\text{-}}) \\ & & \quad +\Sigma\, \tau_1{:}\mathcal{D}(\mathsf{ty}).\, \mathsf{exp}\ (\psi, \mathsf{arr}({\tau_1}^{\text{-}}, \tau)) \times \mathsf{exp}\ (\psi, \tau) \end{array}$$

Note the use of $\tau^{\text{-}}$, discussed above, to flip the polarity of various types. This is necessary because the Σ-bound ty's are bound covariantly, whereas the λ-bound ty is bound contravariantly. An alternative, which discuss below, is to consider a type operator A^{op}, and Σ-quantify over $\mathcal{D}(\mathsf{ty})^{\mathsf{op}}$, in which case τ_1 and τ_2 would be facing the appropriate direction.

Because $\mathsf{ctx} \times \mathcal{D}(\mathsf{ty})$ is a non-dependent pair, there is a transformation $\Psi, \tau \Longrightarrow \Psi', \tau$ whenever there is a transformation $w : \Psi \Longrightarrow \Psi'$. This gives the expected renaming principle: if $e : \mathsf{exp}\ (\psi', \tau)$ then $\mathsf{map}\ (w, \mathsf{refl})\ e : \mathsf{exp}\ (\psi, \tau)$.

Mixed Variance We can also combine dependent types with mixed variance syntax. For example, in Chapter 4, we used a constructor[2]

$\mathsf{dlam} : \{\Gamma : \mathsf{ctx}\}\ \{\mathsf{B} : \mathsf{nat} \to \mathsf{tp}\} \to ((\mathsf{n} : \mathsf{nat}) \to \Gamma \vdash \mathsf{B}\ \mathsf{n}) \to \Gamma \vdash \Pi\ \mathsf{B}$

This constructor can be encoded as follows:

$$\begin{array}{lcl} \vdash & : & (\mathsf{ctx} \times \mathcal{D}(\mathsf{ty})) \to \mathsf{set} \\ \vdash & \cong & \ldots \\ & & \quad +\,\Sigma\, B{:}(\mathsf{nat} \to \mathsf{ty}).\, \mathsf{Id}\ \tau^{\text{-}}\ \mathsf{pi}(B) \times \Pi\, x{:}\mathsf{nat}.\, \vdash (\psi, (B\ x)^{\text{-}}) \\ & & \quad \ldots \end{array}$$

This type is functorial because the index sets of the pi are not parametrized by the context.

[1] Catholic families requires the mysterious equality and its associated miracle of transubstantiation, as the joke goes.

[2] Here we have taken the slight simplification of restricting the index set to nat.

Mixed Variance with Non-constant Domains An example where types appearing in negative positions *are* dependent on a context is Simmons and Toninho (2010)'s constructive provability logic. We encode a simplification here.

CPL offers an explanation of negation-as-failure in logic programming, using a logic that includes reflection on the provability of a statement in another logic. For example, the proposition $\mathsf{sld}(A)$ means that the proposition A can be proved using SLD resolution. More interestingly, the proposition $\mathsf{notsld}(A)$ means that A is not provable using SLD-resolution, and is introduced by giving an admissibility:

$$\frac{(\Gamma \vdash \mathsf{sld}(A)) \to \mathsf{false}}{\Gamma \vdash \mathsf{notsld}(A)}$$

This rule would seem to destroy the structural properties of Γ. However, the truth of $\mathsf{sld}(A)$ depends only on the sld variables in Γ, not ordinary truth variables. Thus Γ is still structural in truth variables.

We can tease this out by separating the definition of the logic into two judgements proves and $\mathsf{provessld}$, and separating Γ into two contexts, one for SLD variables and one for truth variables. $\mathsf{provessld}$ is indexed only by an SLD context, whereas proves is indexed by both contexts. The SLD context appears in both positive and negative positions in proves, so the judgement is *invariant* in the SLD context. However, truth variables occur only positively, so the judgement has the usual structural properties for this context. The type ctx is essentially a higher-dimensional natural number, so an invariant context is just a natural number (with numbers less than it as variables—this type is usually called $\mathsf{fin}(n)$).

The intro rules for notsld and sld are represented as follows:

$$\begin{array}{lcl}
\mathsf{provessld} & : & (\mathsf{nat} \times \mathcal{D}(\mathsf{ty})) \to \mathsf{set} \\
\mathsf{proves} & : & (\mathsf{nat} \times \mathsf{ctx} \times \mathcal{D}(\mathsf{ty})) \to \mathsf{set} \\
\mathsf{proves} & \cong & \lambda(\psi_{\mathsf{sld}}, \psi_{\mathsf{true}}, \phi) \ldots \\
 & & \quad + \Sigma\, \phi' {:} \mathsf{ty}.\, \mathsf{Id}\ \phi^{\text{-}}\ \mathsf{notsld}(\phi') \times (\mathsf{provessld}(\psi_{sld}{}^{\text{-}}, \phi') \to 0) \\
 & & \quad + \Sigma\, \phi' {:} \mathsf{ty}.\, \mathsf{Id}\ \phi^{\text{-}}\ \mathsf{sld}(\phi') \times \mathsf{provessld}(\psi_{sld}, \phi'^{\text{-}}) \\
 & & \quad \ldots
\end{array}$$

Note the use of $-^{\text{-}}$, which is implementable because nat and ty are discrete types, to permit natural numbers and types to be used in mixed variance positions.

The Generic Judgement: First-order logic Next, we consider an example where one index is dependent on another. We can illustrate this with a simple fragment of first-order logic:

$$\frac{\Psi, \mathsf{i} \vdash \phi}{\Psi \vdash \forall\phi}\ \forall I \quad \frac{\Psi \vdash \forall\phi}{\Psi \vdash \phi\{\mathsf{id}, n^{\Psi}\}}\ \forall E$$

For simplicity, we assume that the quantifier can only be instantiated with variables, which means that the elimination rule requires only renaming. There is a renaming operation on derivations as well: if $\mathcal{D} :: \Psi \vdash \phi$ and $r : \Psi \to \Psi'$ then $\mathcal{D}\{r\} :: \Psi' \vdash \phi\{r\}$—the renaming of the derivation proves the renaming of the proposition.

Propositions are a simple syntax:

$$\begin{array}{l}\mathsf{propo} : \mathsf{ctx} \longrightarrow \mathsf{set} \\ \mathsf{propo} \;\cong\; \lambda\,\psi.\,(\mathsf{var}(\psi) + \mathsf{propo}\,(\psi, i))\end{array}$$

We think of the variables as the subjects of a single base proposition $\mathsf{b}(v)$, and the second summand as $\forall$.

The type of natural deduction derivations is defined to satisfy

$$\begin{array}{rcl}\mathsf{nd}\;\Psi\;\phi & \cong & \Sigma\,\phi'{:}\mathsf{propo}\,(\Psi, \mathsf{i}).\,\mathsf{Id}\;\phi\;\forall(\phi') \times \mathsf{nd}\,(\Psi, \mathsf{i})\;\phi' \\ & & +\Sigma\,\phi'{:}\mathsf{propo}\,(\Psi, \mathsf{i}).\,\mathsf{Id}\;\phi\;(\mathsf{map}\;(\mathsf{refl}, v)\;\phi') \times \mathsf{nd}\;\Psi\;\forall(\phi') \\ & & + \ldots\end{array}$$

Representing this parametrization, where both propo and nd are contravariant in Ψ, as a function type requires some additional technology, which we now discuss.

8.2.2 The Attraction of Opposites

In the above examples, we have considered either a hypothetical judgement, or a generic judgement, but not both. An extension is to consider both at once. This would be necessary if, for example, the object logic included both quantifiers and implication. To represent the hypothetical, we use a *type* $\mathsf{hctx}(\Psi)$indexed by a generic context Ψ. However, doing this right requires an $^{\mathsf{op}}$ modality on types.

First, we define a type representing a hypothetical context, which is itself indexed by the generic context Ψ. The reason $\mathsf{hctx}(\Psi)$is a type, rather than set, is that transformations are renamings, analogously to ctx.

$$\frac{\Gamma^{\mathsf{op}} \vdash \Psi : \mathsf{ctx}}{\Gamma \vdash \mathsf{hctx}(\Psi)\;\mathsf{type}} \qquad \frac{}{\Gamma \vdash \epsilon : \mathsf{hctx}(\Psi)} \qquad \frac{\Gamma \vdash \Xi : \mathsf{hctx}(\Psi) \quad \Gamma \vdash \phi : \mathsf{propo}\;\Psi}{\Gamma \vdash \Xi, \phi : \mathsf{hctx}(\Psi)}$$

$$\frac{}{\Gamma \vdash \epsilon : \Xi \Longrightarrow_{\mathsf{hctx}(\Psi)} \epsilon} \qquad \frac{\Gamma \vdash \alpha : \Xi \Longrightarrow_{\mathsf{hctx}(\Psi)} \Xi' \quad \Gamma \vdash v : \mathcal{D}(\phi \in \Xi)}{\Gamma \vdash (\alpha, v) : \Xi \Longrightarrow_{\mathsf{hctx}(\Psi)} \Xi', \phi} \qquad \frac{}{\Gamma \vdash \pi : \Xi, \phi \Longrightarrow_{\mathsf{hctx}(\Psi)} \Xi}$$

$$\frac{\Gamma^{\mathsf{op}} \vdash \Psi : \mathsf{ctx} \quad \Gamma^{\mathsf{op}} \vdash \Xi : \mathsf{hctx}(\Psi) \quad \Gamma \vdash \phi : \mathsf{propo}\;\Psi}{\Gamma \vdash \phi \in \Xi : \mathsf{set}} \qquad \frac{}{\Gamma \vdash \mathsf{z} : \mathcal{D}(\phi \in (\Xi, \phi))} \qquad \frac{\Gamma \vdash v : \mathcal{D}(\phi_0 \in \Xi)}{\Gamma \vdash \mathsf{s}(v) : \mathcal{D}(\phi_0 \in \Xi, \phi)}$$

An object language judgement $\Psi; \Xi \vdash \phi$ will be represented as a type family $\mathsf{nd}\;\Psi\;\Xi\;\phi$. However, we must ensure that nd has the appropriate variance: because of the orientation of the renamings, it should be contravariant in both context arguments. Thus, the obvious candidate type is

$$\Psi : \mathsf{ctx}^-, \Xi : \mathsf{hctx}(\Psi)^-, \phi : \mathsf{propo}\;\Psi^+ \vdash \mathsf{nd}\;\Psi\;\Xi\;\phi\;\mathsf{type}$$

However, this does not type check. The problem is that marking $\Xi : \mathsf{hctx}(\Psi)^-$ as a contravariant assumption means that it should be well-formed contravariantly in the preceding context, whereas in this case it is well-formed covariantly. If we instead mark $\Xi : \mathsf{hctx}(\Psi)^+$ as a covariant

assumption, the context is well-formed, but the type cannot be defined as intended, as the dependence on $\mathsf{hctx}(-)$ faces the wrong direction. An ad-hoc solution to this problem would be to reverse the direction of the morphisms in the definition of $\mathsf{hctx}(-)$.

The right solution is to provide finer control over the variance of types. The essence of the problem is that $\Pi\, x{:}A.\, B$ ties together two independent notions: First, A is well-formed contravariantly in the ambient context. Second, the dependence of B on A is contravariant—for example, in the special case where A and B are both closed, $\Pi\, x{:}A.\, B$ is equivalent to the functor category $A^{\mathsf{op}} \longrightarrow B$.

This limitation can be lifted by introducing a type constructor

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash A^{\mathsf{op}}\ \mathsf{type}}$$

that is interpreted by taking the opposite of A point-wise (i.e. if $A : \Gamma \longrightarrow Cat$ then $A^{\mathsf{op}} = \mathsf{op} \circ A$). We have not yet considered how to present rules for this type syntactically.

nd can then be rendered as

$$\Psi : \mathsf{ctx}^{-}, \Xi : \mathsf{hctx}(\Psi)^{\mathsf{op}^{+}}, \phi : \mathsf{propo}\ \Psi^{+} \vdash \mathsf{nd}\ \Psi\ \Xi\ \phi\ \mathsf{type}$$

This expresses that Ξ is well-formed covariantly in Ψ, but the dependency of nd on it is contravariant.

We can also now explain the dependency of nd as a function type, putting it into the same form as the non-dependent examples discussed above.

$$\mathsf{nd} : (\Sigma\, \Psi{:}\mathsf{ctx}^{\mathsf{op}}.\, \Sigma\, \Xi{:}\mathsf{hctx}(\Psi)^{\mathsf{op}}.\, \mathsf{propo}\ \Psi)^{\mathsf{op}} \to \mathsf{set}$$

The inner op's are necessary for the dependency on Ψ in later components of the Σ; the outer $^{\mathsf{op}}$ cancels the $^{\mathsf{op}}$ that is implicit in $\to$. An alternative would be to use a function space that does not have an implicit $^{\mathsf{op}}$ around the domain, which we discuss below.

Alternative presentation Given the op modality, there is an alternate presentation of 2DTT in which $\Pi\, x{:}A.\, B$ requires a covariant dependence of B on A by default, with contravariant dependency encodable using $^{\mathsf{op}}$. Moreover, this presentation accords slightly better with category-theoretic conventions, in which the Grothendieck construction for a contravariant functor A constructs a fibration $\int_{\Gamma}^{-} A$ as follows:

- an object of $\int_{\mathcal{C}} A$ is a pair (o, a) where $o \in \mathsf{Ob}\ \mathcal{C}$, and $a \in \mathsf{Ob}\ A(o)$.
- a morphism from (o, a) to (o', a') is a pair (c, f) where $c : o \longrightarrow_{\mathcal{C}} o'$ and $f : o \longrightarrow_{A(o')} A(c)o'$

This is related to the covariant Grothendieck construction by

$$(\int_{\Gamma}^{-} A)^{\mathsf{op}} \cong \int_{\Gamma^{\mathsf{op}}} A^{\mathsf{op}}$$

In this notation, our present notion of contravariant context extension $\Gamma\ ,\ x{:}A^{-}$ is interpreted as $\int_{\Gamma}^{-} A^{\mathsf{op}}$, which means that the morphisms require for a map $f : A(c)o' \longrightarrow_{A(o')} o$, facing in the opposite direction from above.

The direct definition of $\int^{-}_{\Gamma} A$ requires the $^{\mathsf{op}}$ modality on types, because the substitution introduction rule is:

$$\frac{\theta : \Gamma \longrightarrow \Delta \quad M : \mathsf{Tm}\ \Gamma^{\mathsf{op}}\ A^{\mathsf{op}}[\theta^{\mathsf{op}}]}{\theta, M : \Gamma \longrightarrow \int^{-}_{\Delta} A}$$

Our current presentation avoids mention of the $^{\mathsf{op}}$ modality by only allowing $\int^{-}_{\Gamma} A^{\mathsf{op}}$, in which case the double-op cancels when the above rule is unfolded.

Of course, given $^{\mathsf{op}}$, this presentation is isomorphic to one that still takes the present definition of $\Gamma\ ,\ x{:}A^{-}$ as primitive, as $\Gamma\ ,\ x{:}A^{\mathsf{op}^{-}}$ interprets as $\int^{-}_{\Gamma} A$ (by canceling double-ops).

8.2.3 Substitution and Other Structural Properties

Above, we have considered only *renamings*, which give weakening, exchange, and contraction for positively occuring variables. Our approach generalizes to additional structural properties by considering additional context categories. For example, the fact that all signatures admit exchange can be expressed by showing that all types are functorial in the category $\mathsf{ctx}^{\cong}$ of contexts and bijections between them. This holds because $\mathsf{ctx}^{\cong}$ has projections both to ctx and $\mathsf{ctx}^{\mathsf{op}}$. The restriction to bijections ensures that inverse exchanges compose to the identity. Similarly, all types admit weakening and strengthening of variables whose types are *insubordinate* to A: all A are functorial in $\mathsf{ctx}^{\geq A}$, the category of contexts that differ only by types that are insubordinate to A, which is again symmetric. This holds because the subordination condition implies that $B \in \Psi$ is preserved by adding or deleting insubordinate A's. These contextual categories can be composed. For example, a schema with positive and negative occurrences of the context is functorial in renamings, which additionally only weaken or contract variables in subordinate to its contravariant types. This can be represented as a category of contexts that has projections to both ctx and $\mathsf{ctx}^{\geq A}$. 2DTT is a natural setting to explore this hierarchy of contexts.

Next, we come to substitution, which has a slightly different flavor. Whereas renaming is functorial, substitution is monadic (Altenkirch and Reus, 1999). More precisely, substitution endows terms with the structure of a *relative monad* (Altenkirch et al., 2010)—which is a generalization of monads to non-endofunctors. This generalization is necessary because $\mathsf{exp}\ \mathsf{ctx} \longrightarrow \mathsf{set}$ is not an endofunctor. Thus, our approach to substitution will be to have a type constructor for constructing a relative monad from a signature.

In simpler terms, the programmer will write a signature expressed as a (perhaps strictly positive) functor that is parametrized by the recursive call. E.g.

$$\mathsf{exp} : (\mathsf{ctx} \longrightarrow \mathsf{set})^{+}, \psi : \mathsf{ctx}^{-} \vdash expsig : \mathsf{set}$$
$$expsig = (\mathsf{exp}\ \psi \times \mathsf{exp}\ \psi) + \mathsf{exp}\ (\psi, i)$$

From this we can generate

- The type $\mathsf{exp} : \mathsf{ctx} \longrightarrow \mathsf{set} = \lambda\, \psi.\, \mu(\mathsf{exp}.\psi.expsig, \psi)$
- The type ctx' of term-for-variables substitutions.
- The type $\mathsf{exp}' : \mathsf{ctx}' \longrightarrow \mathsf{set}$ of expressions equipped with substitution.

Given a signature that potentially takes advantage of dependent types and mixed variance, this construction will allow the programmer to get all of the available structural properties for free from the signature description.

8.2.4 Covariant Π's

Using the rules above, a contravariant assumption can be internalized as a function: if $\Gamma, x : A^- \vdash M : B$ then $\Gamma \vdash \lambda\, x.\, M : \Pi\, x{:}A.\, B$. It is natural to ask whether anything analogous can be said for covariant variables: is there any sort of function space internalizing $\Gamma, x : A^+ \vdash M : B$? As we have observed above, the problem with attempting to naïvely construct such a function space is that A is well-formed covariantly in Γ, so map is not definable by pre- and post-composition.

However, there is in fact such a function type, and, as it turns out, we have already seen the seeds of it in Chapter 5: In the NBE example, we observed that a function of type

$$\forall \Psi_1.\langle \Psi + \Psi_1 \rangle \mathsf{sem} \to \langle \Psi + \Psi_1 \rangle \mathsf{sem}$$

is always weakenable to any context bigger than Ψ, because it explicitly accounts for such extensions. Thus, map is definable by instantiating the quantifier.

Categorically, this type corresponds to the exponential object in the category of presheaves: given two functors $A, B : \Gamma \to Sets$, where Γ is a small category, there is a functor $B^A : \Gamma \to Sets$ defined by

$$\begin{array}{rcl} B^A(o) & = & Hom_{Sets^\Gamma}(Hom_\Gamma(o,-) \times A, B) \\ B^A(c : o1 \longrightarrow_\Gamma o2) & = & f \mapsto (o' \mapsto \ \ (c' : o_2 \longrightarrow o', x) \mapsto f_{o'}(c' \circ c, x)) \end{array}$$

Here $Hom_\Gamma(o,-) : \Gamma \longrightarrow Sets$ is the covariant hom functor, which sends each object o' to the set of maps in Γ from o to o':

$$\begin{array}{rcl} Hom(o,-)(o') & = & Hom_\Gamma(o,o') \\ Hom(o,-)(c : o_1 \longrightarrow_\Gamma o_2) & = & (f : o \longrightarrow o_1) \mapsto c \circ f \end{array}$$

Thus, B^A sends each object o to the set of natural transformations (which are the maps in the functor category $Sets^\Gamma$) from $Hom_\Gamma(o,-) \times A$ to B. Spelling out this definition, we see that an element of $B^A(o)$ is a function $\forall o'.Hom_\Gamma(o,o') \times A(o') \to B(o')$, which is natural in o'. If we ignore the naturality constraint, this specializes to the type above if we take $Hom(\Psi, \Psi')$ to mean $\Psi' = \Psi + \Psi_1$ for some Ψ_1.

The action on morphisms works by instantiating the quantifier: if we are given $f : Hom_\Gamma(o_1,-) \times A \Longrightarrow B$ and $(c : o_1 \longrightarrow_\Gamma o_2)$, we need to make a natural transformation $g : Hom_\Gamma(o_2,-) \times A \Longrightarrow B$. This is defined by taking

$$g_{o'} = (c' : o_2 \longrightarrow o', x : A(o)) \mapsto f_{o'}(c' \circ c, x)$$

Naturality follows from the naturality of f. That is, f already works for all "future" objects of Γ, and by composition the future of the future is the future.

Our next task is to figure out how to describe this type syntactically. A first cut is to define a type

$$\frac{\Gamma \vdash A : \mathsf{set} \quad \Gamma \vdash B : \mathsf{set}}{\Gamma \vdash B^A : \mathsf{set}}$$

However, there are two problems with this rule. The first is technical: the object $B^A ofSets^\Delta$ does not interact in the usual way with substitution, in that $B^A[\theta]$ is different than $B[\theta]^{A[\theta]}$. For an

(Non-dependent) context append:

$$\frac{\Gamma\ \mathsf{ctx} \quad \Delta\ \mathsf{ctx}}{\Gamma, \Delta\ \mathsf{ctx}} \qquad \frac{\Gamma \vdash \theta_1 : \Delta_1 \quad \Gamma \vdash \theta_2 : \Delta_2}{\Gamma \vdash \theta_1, \theta_2 : \Delta_1, \Delta_2} \qquad \frac{\Gamma \vdash \delta_1 : \theta_1 \Longrightarrow_{\Delta_1} \theta_1' \quad \Gamma \vdash \delta_2 : \theta_2 \Longrightarrow_{\Delta_2} \theta_2'}{\Gamma \vdash (\delta_1, \delta_2) : \theta_1, \theta_2 \Longrightarrow_{\Delta_1, \Delta_2} \theta_1', \theta_2'}$$

Non-dependent transformation assumptions:

$$\frac{\Gamma\ \mathsf{ctx} \quad \Delta\ \mathsf{ctx} \quad \Gamma \vdash \theta_1 : \Delta \quad \Gamma \vdash \theta_2 : \Delta}{\Gamma\ ,\ d{:}\theta_1 \Longrightarrow_\Delta \theta_2\ \mathsf{ctx}} \qquad \frac{d : \theta_1 \Longrightarrow_\Delta \theta_2 \in \Gamma}{\Gamma \vdash d : \theta_1 \Longrightarrow_\Delta \theta_2} \qquad \frac{\Gamma \vdash \theta : \Delta \quad \Gamma \vdash \delta : \theta_1[\theta] \Longrightarrow_\Psi \theta_2[\theta]}{\Gamma \vdash \theta, \delta/d : \Delta, d : \theta_1 \Longrightarrow_\Psi \theta_2}$$

$$\frac{\begin{array}{l}\Gamma \vdash \delta_0 : \theta \Longrightarrow_\Delta \theta' \\ \Gamma \vdash \mathsf{refl}_{\theta_2}[\delta_0] \circ \delta \equiv \delta' \circ \mathsf{refl}_{\theta_1}[\delta_0] : \theta_1[\theta] \Longrightarrow_\Psi \theta_2[\theta']\end{array}}{\Gamma \vdash (\delta_0, \star) : \theta, \delta \Longrightarrow_{\Delta, d:\theta_1 \Longrightarrow_\Psi \theta_2} \theta', \delta'}$$

Covariant functions:

$$\frac{\Delta\ \mathsf{ctx} \quad \Gamma \vdash \theta : \Delta \quad \Delta \vdash A, B : \mathsf{set}}{\Gamma \vdash \Pi^+(\Delta \geq \theta).A \to B : \mathsf{set}} \qquad \frac{\Gamma, \Delta, d : (\theta \Longrightarrow_\Delta \mathsf{id}_\Delta), x : \mathcal{D}(A)^+ \vdash M : \mathcal{D}(B)}{\Gamma \vdash \lambda\Delta, d, x.M : \mathcal{D}(\Pi^+(\Delta \geq \theta).A \to B)}$$

$$\frac{\Gamma \vdash M : \mathcal{D}(\Pi^+(\Delta \geq \theta_1).A \to B) \quad \Gamma \vdash \theta_2 : \Delta \quad \Gamma \vdash \delta : \theta_1 \Longrightarrow_\Delta \theta_2 \quad \Gamma \vdash N : \mathcal{D}(A[\theta_2])}{\Gamma \vdash M(\theta_2, \delta, N) : \mathcal{D}(B[\theta_2])}$$

$$\begin{array}{rcll}
(\lambda\Delta, d, x.M)(\theta_2, \delta, N) & \equiv & M[\mathsf{id}_\Gamma, \theta_2, \delta/d, N^+/x] & \beta \\
M & \equiv & \lambda\Delta, d, x.M(\mathsf{id}_\Delta, d, x) & \eta \\
 & & & \\
(\Pi^+(\Delta \geq \theta).A \to B)[\theta_0] & \equiv & \Pi^+(\Delta \geq \theta[\theta_0]).A \to B & \textit{1-subst} \\
(\Pi^+(\Delta \geq \theta).A \to B)[\delta] & \equiv & f.\lambda\Delta, d, x.f(\mathsf{id}_\Delta, d \circ \theta[\delta], x) & \textit{1-resp} \\
 & & & \\
(\lambda\Delta, d, x.M)[\theta_0] & \equiv & \lambda\Delta, d, x.M[\theta_0, \mathsf{id}\Delta, d/d, x^+/x] & \textit{1-subst} \\
M(\theta_2, \delta, N)[\theta_0] & \equiv & M[\theta_0](\theta[\theta_0], \delta[\mathsf{refl}_{\theta_0}], N[\theta_0]) & \\
(\lambda\Delta, d, x.M)[\delta] & \equiv & \star & \textit{1-resp} \\
M(\theta_2, \delta, N)[\delta] & \equiv & \star &
\end{array}$$

Figure 8.2: 2DTT: Covariant Functions

object $o \in \Delta$, the former quantifies over all futures of $\theta(o)$ in Γ, whereas the latter quantifies over all futures of o in Δ, and these are not necessarily the same. Thus, we cannot define substitution compositionally. The second is practical: the type B^A doesn't quite match our intuition from the NBE example, as there is no place to write the bounding context Ψ.

Both of these problems arise because the type $\Gamma \vdash B^A : \mathsf{set}$ is defined so that it anticipates extensions *in the ambient context* Γ, which means that this type reflects upon the context in a way that other type constructors do not. In type theory, the connectives always build in a composition, so that they can be judged well-formed in an arbitrary context. The above rule fails to do so.

The solution is to build in this composition and make the bound an explicit part of the type constructor: we define a type $\Pi^+(\Delta \geq \theta).A \to B$

$$\frac{\Delta\ \mathsf{ctx} \quad \Gamma \vdash \theta : \Delta \quad \Delta \vdash A, B : \mathsf{set}}{\Gamma \vdash \Pi^+(\Delta \geq \theta).A \to B : \mathsf{set}}$$

For example, if we take Δ to be a single assumption $\Psi' : \mathsf{ctx}$, and Ψ is a context, and A, B are well-formed in $\Psi' : \mathsf{ctx}$, then we can form the type $\Pi^+(\Psi' : \mathsf{ctx} \geq \Psi).A(\Psi') \to B(\Psi')$, as above. Then the substitution into $\Pi^+(\Delta \geq \theta).A \to B$ is defined by composing substitutions.

Next, we need intro and elim rules for this connective. Based on the semantics, $\Pi^+(\Delta \geq \theta).A \to B$ should classify functions, that, for every θ' reachable from θ and $A[\theta]$, produce a $B[\theta']$. Stating this syntactically requires a couple of new ingredients: substitution and transformation variables. Thus far, we have considered the transformation judgements only on the right. It is also possible to allow *hypothetical transformations* as a judgemental notion, even though it is not obvious how to internalize transformations as a dependent type.

In Figure 8.2, we give the rules for these concepts: The first set of rules allow a context to be extended with an entire context (which is not dependent on the first)—the rules are exactly those for the product category. The next set of rules define transformation variables: a context can be extended with a transformation variable, and transformation variables can be used as transformations. A substitution into $\Gamma\ ,\ d{:}\theta_1 \Longrightarrow_\Delta \theta_2$ consists of a substitution into Γ and a transformation for the substitution instance. A transformation *between* such substitutions consists of a transformation for the first component, such that a naturality condition holds for the second components—because the theory is 2-dimensional, there is no notion of morphism between transformations. We elide the equality rules for these contexts. We could also allow term transformation variables $a : M \Longrightarrow_A N$, but they are not necessary for this construction.

The introduction rule for $\Pi^+(\Delta \geq \theta).A \to B$ internalizes a hypothetical term as a function. Note the weakenings: θ is dependent only on Γ, and id_Δ and A and B only on Δ. The elimination rule is symmetric. The $\beta\eta$ rules express that this is an isomorphism. The 1-resp rule for $\Pi^+(\Delta \geq \theta).A \to B$ simply composes substitutions. The 1-resp rule composes transformations (the future of the future is the future), as discussed above. It is also necessary to adjust Δ by θ because of the built-in composition. The 1-subst rules are compositional; the 1-resp rules are computationally trivial, but require checking that the equations hold.

While these rules are operationally sensible, they allow impredicativity. For example, Δ might be $a : \mathsf{set}$, in which case $\Pi^+(a : \mathsf{set} \geq 0^+/a).A \to B$ permits impredicative polymorphic quantification over sets, as in the System F type $\forall a.A(a) \to B(a)$. Thus, we may wish to restrict the formation rule for covariant Π's in order to avoid impredicativity. We leave an investigation

of the necessary restrictions to future work; for example, quantification over set might increase the size of the type.

The type $\Pi^+(\Delta \geq \theta).A \to B$ described here can be generalized in two ways: First, we can generalize the $\to$ to a Π, by allowing the domain type type A to depend on $d : \theta \Longrightarrow \mathsf{id}_\Delta$, and the range type B to depend on both d and $x : A^+$. Second, we can generalize from sets to types. Here, we have considered the exponential of two presheaves. This construction generalizes to functors into Cat, using the exponential of two fibrations.

Applications at set If Δ has an initial object, then we can derive an unbounded quantifier from the bounded quantifier. For example, at set, we write $\Pi^+(a : \mathsf{set}).A \to B$ for $\Pi^+(a : \mathsf{set} \geq 0^+/a).A \to B$. There is a unique transformation $0^+ \Longrightarrow_{\mathsf{set}} S^+$ given by $x.\mathsf{abort}\, x$, so we can ignore the transformation argument to functions and applications.

Naturality properties of polymorphic functions are typically only made available in external logics based on relational parametricity. In 2DTT, they are instead available as definitional equality principles inside the type theory. For example, we can derive the fact that a polymorphic function on lists, of type $\Pi^+(a : \mathsf{set}).\mathsf{list}(a) \to \mathsf{list}(a)$, commutes with map:

$$\begin{array}{ll} \text{If} & P : \Pi^+(a : \mathsf{set}).\mathsf{list}(a) \to \mathsf{list}(a) \\ & x.F : S_1 \Longrightarrow_{\mathsf{set}} S_2 \\ & L : \mathsf{list}(S_1) \\ \text{then} & \mathsf{map}_{a.\,\mathcal{D}(\mathsf{list}(a))}\ (x.F)\ P(S_1, L) \equiv P(S_2, \mathsf{map}_{a.\,\mathcal{D}(\mathsf{list}(a))}\ (x.F)\ L) \end{array}$$

This makes essential use of directedness: in a higher-dimensional symmetric type theory, $S_1 \Longrightarrow_{\mathsf{set}} S_2$ would classify only equivalences, though typical applications of this reasoning principle apply it to arbitrary functions from S_1 to S_2.

Contravariant $\Pi\, x{:}A.\, B$ satisfies naturality conditions as well. The reason it cannot be used for this example is that the type $\Pi\, a{:}\mathsf{set}.\, \mathsf{list}(a) \to \mathsf{list}(a)$ is not well-formed, because the assumption a can only be used in contravariant positions. The type $\Pi^+(a : \mathsf{set}).\mathsf{list}(a) \to \mathsf{list}(a)$ is necessary to allow a to be used in both the domain and range.

Applications at ctx Using $\Pi^+(\psi : \mathsf{ctx}^- \geq \Psi^-/\psi).\mathsf{exp}\ \psi \to \mathsf{exp}\ \psi$ we can reproduce some familiar results relating functions to de Bruijn form, which has applications to higher-order abstract syntax (Fiore et al., 1999; Hofmann, 1999).

Suppose ctx includes term-for-variable substitutions. Then there is a type isomorphism

$$(\Pi^+(\psi : \mathsf{ctx}^- \geq \Psi^-/\psi).\mathsf{exp}\ \psi \to \mathsf{exp}\ \psi) \cong \mathsf{exp}\ (\Psi, \mathsf{i})$$

Intuitively, this says that the space of functions that work in all future contexts, *naturally in substitutions*, is the same as the space of derivabilities represented in de Bruijn form. The reason for this is that both represent exactly substitution functions. Thus, we have a *negative* (in the focusing sense) function space that is isomorphic to a *positive* representation in de Bruijn form. This provides another explanation as to why substitution functions can be used in inductive definitions, and why their polarity is somewhat confusing: they can be isomorphically thought of as either positive or negative.

From right to left, the isomorphism sends

$$e \mapsto \lambda\psi, d, x.\mathsf{map}_{\psi.\mathsf{exp}\,\psi}\ (d, x)\ e$$

which substitutes its argument for the last variable in e. From left to right, it instantiates the quantifier with a context with one extra variable, plugs π in for the transformation, and plugs the new variable in for x.

$$f \mapsto f((\Psi, \mathsf{i})/\psi, \pi, \mathsf{v}(\mathsf{z}))$$

Using naturality, these maps can be shown to be mutually inverse. Without naturality, these maps back and forth can be defined, but the type $(\Pi^{+}(\psi : \mathsf{ctx}^{-} \geq \Psi^{-}/\psi).\mathsf{exp}\ \psi \to \mathsf{exp}\ \psi)$ is broader than $\mathsf{exp}\ (\Psi, \mathsf{i})$, because its elements functions may analyze their argument.

HOAS At first glance, one might hope that this type isomorphism will provide a convenient higher-order way to write down syntax. Unfortunately, elements of $(\Pi^{+}(\psi : \mathsf{ctx}^{-} \geq \Psi^{-}/\psi).\mathsf{exp}\ \psi \to \mathsf{exp}\ \psi)$ are no easier to write than de Bruijn terms: as the above isomorphism shows, a term $\lambda\psi', d, x.e$ gets a new weakening function (d) and a new last variable (x), and weakening and other structural properties must be marked explicitly, just as in de Bruijn form.

However, following Hofmann (1999), we may be able to introduce HOAS as a special syntax for the functor category $\mathsf{ctx} \to \mathsf{set}$. This category is itself a Cartesian closed category, with exponentials B^A given by $\lambda\,\psi.\,(\Pi^{+}(\psi' : \mathsf{ctx}^{-} \geq \psi^{-}/\psi').A\ \psi' \to B\ \psi')$. The interpretation of λ-terms into this CCC thus constitutes a semantically motivated parser trick for writing de Bruijn form in a higher-order notation.

8.2.5 Higher-Dimensional Quotient Types

In ordinary type theory, one can consider quotient types A/R, where R is an equivalence relation on A: the elements of A/R are the elements of A, but two elements are considered equal whenever they are related by R. In 2DTT, the analogous operation is forming a type from an *internal category*—a description of a category inside the theory. In the symmetric higher-dimensional case, we can introduce a new type-forming operation as follows:

$$\frac{\begin{array}{l} O : \mathsf{set}^{\cong} \\ A : O \to O \to \mathsf{set}^{\cong} \\ r : \Pi\, x{:}O.\ A\ x\ x \\ t : \Pi\, x_1, x_2, x_3{:}O.\ A\ x_2\ x_3 \to A\ x_1\ x_2 \to A\ x_1\ x_3 \\ t\ r\ p \equiv p \\ t\ p\ r \equiv p \\ t(t\ p_1\ p_2)p_3 \equiv t\ p_1\ (t\ p_2\ p_3) \end{array}}{\mathsf{cat}(O, A, r, t)\ \mathsf{type}}$$

Terms and equivalences are introduced by the corresponding sets:

$$\frac{M : O}{[M] : \mathsf{cat}(O, A, r, t)} \qquad \frac{P : A\ [M_1]\ [M_2]}{[P] : \mathsf{Id}_{\mathsf{cat}(O,A,r,t)}\ [M_1]\ [M_2]}$$

and terms can be eliminated by a raw function that respects equality:

$$\frac{M : \mathsf{cat}(O, A, r, t) \quad R : O \to C \quad P : \Pi\, x, y{:}O.\, A\ x\ y \to \mathsf{Id}_C\ (R\ x)\ (R\ y)}{\mathsf{celim}(M, R, P) : C}$$

Adapting these rules to the directed case is an important piece of future work, as it will allow programmers to introduce types such as ctx and $\mathsf{hctx}(\Psi)$, and write elimination forms such as π, without extending the language.

Discretization We can go further and expose the fact that *every* type can be viewed in the form $\mathsf{cat}(O, A, r, t)$. To do this, we need an operation $!A$ that forgets the higher-dimensional structure of a type. Above, we considered nat to be an "invariant" version of ctx. This is an instance of a general construction: Given a structured type A, there is a set $\mathsf{Ob}\ A$, and therefore a type $\mathcal{D}(\mathsf{Ob}\ A)$. This type forgets the morphism part of A, leaving only the underlying set of objects, which are now only transformable if they are equal. In fact, $\mathsf{Ob}\ -$ and $\mathcal{D}(-)$ are a section-retraction, because $\mathsf{Ob}\ \mathcal{D}(S) \equiv S$. Semantically, it is clear what the functor $\mathsf{Ob}\ : Cat \longrightarrow Sets$ means, but we have not investigated a proof theory for it. Because $\mathcal{D}(\mathsf{Ob}\ A)$ forms a comonad, which we will abbreviate as $!A$, it could potentially be given a proof theory analogous to the necessitation operator in modal logic (Pfenning and Davies, 2001). For example, there is an operation $\mathsf{un!}\ M$ which has type A if $M : !A$.

Using !, we can directly expose the semantic definitions of types and terms in the type theory. For example, we may add a rule that encodes the definition of $\mathsf{Tm}\ \Gamma\ A$:

$$\frac{\begin{array}{l}\Gamma \vdash A\ \mathsf{type} \\ !\Gamma \vdash M : !A[\mathsf{un!\ id}] \\ \vec{x} :!\Gamma, \vec{y} :!\Gamma, d : \mathsf{un!}\ \vec{x} \Longrightarrow_A \mathsf{un!}\ \vec{y} \vdash \alpha : \mathsf{un!}\ (\mathsf{map}_A\ d\ M[\mathsf{id}_{\vec{x}}]) \Longrightarrow_{A[\mathsf{un!}\ \vec{y}]} \mathsf{un!}\ (M[\mathsf{id}_{\vec{y}}]) \\ \alpha[\mathsf{refl}] \equiv \mathsf{refl} \\ \alpha[d_2 \circ d_1] \equiv \alpha[d_2/d] \circ \mathsf{resp}^1\ (x.\mathsf{map}\ d_2\ x)\ \alpha[d_1/d]\end{array}}{\Gamma \vdash \mathsf{explicit}(M, \alpha) : A}$$

8.2.6 Directed Hom Types

Thus far, we have treated equality as a judgement, which is necessary because the usual construction of the identity type in symmetric type theory does not apply in the directed case. For review, identity types

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Id}_A\ M\ N : \mathsf{set}}$$

are interpreted by

$$(\mathsf{Id}_A\ M\ N)(o) = Hom_{A(o)}(M(o), N(o))$$
$$(\mathsf{Id}_A\ M\ N)(c : o_1 \longrightarrow o_2) = f : Hom_{A(o_1)}(M(o_1), N(o_1)) \mapsto N(c) \circ A(c)(f) \circ M(c)^{-1}$$

The functoriality of the type constructor depends on the symmetry of $Hom_{A(o)}$, because $M(c) : A(c)M(o_1) \longrightarrow M(o_2)$, but the construction requires a map $M(c) : M(o_2) \longrightarrow A(c)M(o_1)$.

In addition to the judgemental notion of transformation, it would be useful to have a first-class *directed hom type*, which could be freely mixed with the other type constructors. There are several potential solutions to this problem:

One option is to define the hom-types a relation on *objects* of a type, using the ! type constructor discussed above:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash M : !A \quad \Gamma \vdash N : !A}{\Gamma \vdash \mathsf{hom}_A\ M\ N : \mathsf{set}}$$

Transformation at $!A$ is just equality, and it is therefore symmetric even if A is not. This definition is restrictive, because equations are restricted to elements M that respect transformation on the context on-the-nose, rather than up to transformation. However, it matches category-theory practice of defining the Hom-set on the objects of a category.

Another solution is to define the hom-type so that it is explicitly contravariant in M, which accounts for the symmetry required by functoriality.

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash M : A^{\mathsf{op}} \quad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{hom}_A\ M\ N : \mathsf{set}}$$

However, it is unclear what the rules for this type should be—for example, reflexivity and transitivity do not (obviously) make sense, because it relates two terms of opposite variances.

Both of these types make sense semantically, but it is unclear what the right notion to integrate into the type theory is, or how they related to each other and to the judgemental notion of transformation.

8.2.7 Universes and Higher-dimensions

Existing proof assistants have a *size* hierarchy which is used to avoided the paradox of type : type. Higher-dimensional type theories have an additional axis of *dimensionality*. These are mostly orthogonal ideas: there can be small types of high dimension, and large types of low dimension. However, there are some relationships; for example, the collection of all d-dimensional types of size l naturally forms a $n+1$-dimensional type of size $l+1$. For example, the collection of all sets naturally forms a large category. Exploring this matrix is an important piece of future work, generalizing 2DTTto ωDTT. Here, we take the first step of situating our present approach in the hierarchy. For simplicity, we discuss these issues in the symmetric setting.

Current proof assistants such as Agda and NuPRL commit to the fact that all types are sets, in the above sense of being of dimension one. However, intensional type theories, like Agda, and extensional type theories, like NuPRL, differ in how they expose this set-hood to the user. In intensional type theories like Agda, it is through *propositional uniqueness of identity proofs*. This can be expressed in many equivalent ways (see Hofmann and Streicher (1998)), one of which is that all proofs of propositional equality are themselves propositionally equal: for all A, there is a term of type UIPA, where

$\mathsf{UIP}\ \mathsf{A} = \{\mathsf{x}\ \mathsf{y} : \mathsf{A}\}\ \{\mathsf{p}\ \mathsf{q} : \mathsf{Id}\ \mathsf{A}\ \mathsf{x}\ \mathsf{y}\} \to \mathsf{Id}\ (\mathsf{Id}\ \mathsf{A}\ \mathsf{x}\ \mathsf{y})\ \mathsf{p}\ \mathsf{q}$

On the other hand, in extensional type theories, set-hood is expressed in two ways: (1) propositional equalities induce definitional equalities and (2) all proofs of propositional equality are

definitionally equal:

$$\frac{P : \mathsf{Id}_A\ M\ N}{M \equiv N : A}\ \textit{reflect} \qquad \frac{P : \mathsf{Id}_A\ M\ N}{P \equiv \mathsf{refl}_M : \mathsf{Id}_A\ M\ N}\ \textit{uip-def}$$

Note that the first rule is necessary for type-checking the second.

This raises the question of how these notions of intensional and extensional set-hood generalize to higher-dimensional type theories. Here, we argue that they arise from two different interpretations of the definition of an n-groupoid. Consider a general ω-groupoid interpretation of type theory (Garner, 2009; Lumsdaine, 2009). Semantically, a closed type A denotes an ω-groupoid. By definition an ω-groupoid is an n-groupoid if, for $j > n$, every two parallel j-morphisms are equivalent. This expresses a *cut-off condition* which says that the structure above dimension n is trivial.

Intensional n-groupoids arise from transcribing this cut-off condition directly into the type theory, using propositional equality for equivalence:

DEFINITION 8.2.1: INTENSIONAL n-GROUPOIDS.

1. Intensional Propositions: A is an intensional -1-groupoid if A is an intensional 0-groupoid and there is a term of type
   `(x y : A) -> Id A x y`
2. Intensional Sets: A is an intensional 0-groupoid if A is an intensional 1-groupoid and there is a term of type
   `(x y : A) (p q : Id A x y) -> Id (Id A x y) p q`
3. Intensional Groupoids: A is an intensional 1-groupoid if A is an intensional 2-groupoid and there is a term of type
   `(x y : A) (p q : Id A x y) (r s : Id (Id A x y) p q) -> Id r s`
4. ...
5. A is an intensional ω-groupoid

These definitions say that, *up to propositional equality*, a proposition has only one inhabitant, a set is discrete, the 2-cells of a groupoid are discrete, etc.

On the other hand, we can ask for the stronger condition that this cut-off condition holds up to equality, not just equivalence: A is *strictly* an n-category[3] if for all $j > n$, every j-morphism is the identity.

Rendering this condition type-theoretically gives the rules of extensional type theory:

DEFINITION 8.2.2: EXTENSIONAL n-GROUPOIDS.

1. Extensional Propositions: A is an extensional -1-groupoid if A is a 0-groupoid and for all $M, N : A$, $M \equiv N$ (the set of 0-morphisms from M to N is defined to be the one-element set). I.e.
$$\frac{}{M \equiv N : A}\ \textit{reflect}$$

[3]Note that "strictly" has nothing to do with whether the n-category is strict or weak—whether the identity and composition laws hold on the nose at each level, or up to higher-dimensional structure—but with whether the cut-off condition is phrased in terms of equality or equivalence.

2. Extensional Sets: A is an extensional 0-groupoid if A is a 1-groupoid and for all $M, N : A$ if $P : \mathsf{Id}_A\ M\ N$ then $M \equiv N$ and $P \equiv \mathsf{refl}_M$:

$$\frac{P : \mathsf{Id}_A\ M\ N}{M \equiv N : A}\ \textit{reflect} \qquad \frac{P : \mathsf{Id}_A\ M\ N}{P \equiv \mathsf{refl}_M : \mathsf{Id}_A\ M\ N}\ \textit{uip-def}$$

3. Extensional Groupoids: A is an extensional 2-groupoid if A is a 2-groupoid and

$$\frac{R : \mathsf{Id}_{\mathsf{Id}_A\ M\ N}\ P\ Q}{P \equiv Q : \mathsf{Id}_A\ M\ N}\ \textit{reflect} \qquad \frac{R : \mathsf{Id}_{\mathsf{Id}_A\ M\ N}\ P\ Q}{R \equiv \mathsf{refl}_P}\ \textit{uip-def}$$

4. ...
5. A is an extensional ω-groupoid

Note that this definition uses definitional equality—i.e. equality on objects—and is hence "evil" (not stable under equivalence). In the limit, the intensional and extensional ω-groupoids are the same—because there is no cutoff-condition.

The usual trade-offs between intensional and extensional type theory apply in the higher-dimensional case: Extensional groupoids make type checking a term undecidable, because *reflect* throws away the equality proof. Additionally, in extensional type theory it is not possible to implement equality on open-terms by untyped reduction, as in the presence of contradictory assumptions, all terms are equal, and hence terms that would ordinarily be ill-typed are well-typed (e.g. $\mathsf{int} \equiv \mathsf{int} \times \mathsf{int}$, so $\mathsf{fst}\ 6$ is well-typed). This problem can be solved by defining reduction on ill-typed terms, as in NuPRL's direct computation (Constable et al., 1986). On the other hand, the extensional theory is more general, because it can describe both extensional and intensional groupoids. Intensional type theory restores decidability by forcing one to always work up to equivalence, never equality. For example, if nat is defined as an intensional set by the usual inductive definition, then any type family $x : \mathsf{nat} \vdash C\ \mathsf{type}$ has an action $\mathsf{map}_C\ (P : \mathsf{Id}_{\mathsf{nat}}\ M\ N)\ : C[M] \cong C[N]$. In fact, this isomorphism is the identity, but intensional type theory assumes the worst—that it is an arbitrary isomorphism—and forces the programmer to push it around a term explicitly—e.g., canceling inverses, or substituting P with another proof Q. Thus, extensional type theory can be more convenient to use, because coercions by *equalities* (as opposed to more general equivalences) are tacit, and thus there is never a need to reason *about* them.

OTT (Altenkirch et al., 2007) combines intensional type theory (no equality reflection rule) with extensional concepts such as proof-irrelevance and functional extensionality, all without introducing stuck closed terms (a problem simple axiomatic treatments of functional extensionality suffer from). It does this by distinguishing a universe of proof-irrelevant propositions prop from arbitrary sets. Under the present analysis, we see that an OTT proposition is an *extensional proposition*: all proofs of it are *definitionally* equal. On the other hand, an OTT set is an *intensional set* (intensional 0-groupoid): while all proofs of propositional equality are propositionally equal (*uip-def*), propositional equality does not induce a definitional equality (*reflect*). That said, on OTT set is trivially an *extensional 1-groupoid*, because all proofs of higher dimensional identities are definitionally equal. This corresponds to the original semantic motivation for OTT (Altenkirch, 1999), as an intensional set that is also an extensional groupoid is exactly a setoid.

Adapting this terminology to the directed case, we say that 2DTT has a universe of extensional 1-categories ($\Gamma \vdash A\ \mathsf{type}$), and a universe of extensional 0-categories ($\Gamma \vdash S : \mathsf{set}$), and nothing else. The higher-dimensional and intensional variants are an interesting subject for future work.

8.3 Related Work

The functorial/monadic approach to abstract syntax was developed by Altenkirch and Reus (1999); Fiore et al. (1999); Hofmann (1999). 2DTT allows this theory to easily be put into practice, by providing a language in which it is easy to describe categorical constructions. Additionally, the signatures available in 2DTT are more general, in that they allow mixing of admissibility and derivability—existing functorial accounts of syntax concentrate only on derivability. Fiore's later work (Fiore, 2008) gives a semantics for second-order and dependently typed abstract syntax, though not both at once. To our knowledge, the analysis of the structural properties of the generic judgement in terms of the Grothendieck construction is novel, or at least has not been used in type theory. 2DTT will enable us to go beyond LF-based systems (Altenkirch and Reus, 1999; Pfenning and Schürmann, 1999; Pientka, 2008; Poswolsky and Schürmann, 2008) by supporting both admissibility and derivability.

Pouillard and Pottier (2010) describe an interface to dependent de Bruijn indices that ensures that any function $\Pi\,\psi{:}\mathsf{ctx}.\,A$ commutes with bijections on variables. 2DTT improves on this in several ways: First, we generalize this result to other notions of transformation, including directed ones: ctx can be chosen to force such a function to respect only equality, bijections, renamings, or general substitutions. Second, Pouillard and Pottier (2010)'s representation is somewhat non-standard: contexts are treated as abstract type, and de Bruijn indices are represented relationally, by defining a relation that means i is the successor of j. This allows naturality at ctx to be encoded using the naturality of abstract types. In 2DTT, we do not need this encoding. Third, in Pouillard and Pottier (2010), these naturality properties are not available for reasoning inside of of the type theory, only in an external relational interpretation. In contrast, 2DTT treats them as rules of the type theory.

Chapter 9

Conclusion

Any stamp collector will recognize the Inverted Jenny, a 1918 24-cent misprint featuring an upside-down Curtiss JN-4 airplane. In a sense, the work described in this dissertation is also upside-down, compared to the textbook dissertation: we begin in Part I with the most practical work, and get progressively more theoretical in Parts II and III. In Part I, we argued that it is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language, by showing how to construct a security-typed language and a semantic type system for differential privacy. These examples make essential use of both derivability and admissibility, and the remainder of the thesis describes progress towards putting these two notions of consequence on equal footing, rather than privileging one over the other (derivability in LF, admissibility in MLTT). In Part II, we described a logical framework that allows derivability and admissibility hypothetical judgements to be mixed in novel and interesting ways. The fact that admissibilities can invalidate the structural properties of derivability was handled by a collection of ad-hoc tactics that implement the structural properties in many useful cases. In Part III, we analyzed these tactics as functoriality, and showed that a directed type theory, inspired by higher category theory, accounts for the structural properties of the generic judgement. Our investigation into programming with logics prompted two independently interesting technical contributions, spanning all three points of the Curry-Howard correspondence. The first, higher-order focusing for intuitionistic logic, is a logical formalism that handles inductive types well, using an infinitary proof theory. The second, directed type theory, is an exciting and fundamental generalization of dependent type theory that accounts for asymmetric concepts.

Thus, the real test of this work will be whether we can, in the next several years, bring it back down to Earth. What is the road map to doing so? The first step is the theoretical work on directed type theory outlined above:

1. It is paramount to adapt familiar type constructors, such as an inductive datatype mechanism, to the directed case. Inductive types may be handled by giving a directed analogue of W-types (Nordström et al., 1990) or indexed containers (Altenkirch and Morris, 2009).
2. The analog of symmetric type theory's quotient types is the notion of an internal category in directed type theory, which will allow programmers to define their own directed types by giving a collection of elements and a notion of transformation between them.
3. The concept of the opposite of a category, which ordinarily plays a central role in account-

ing for variances, is much more subtle in the dependent case than in the non-dependent case usually considered. A full accounting of variances involves a modality for contravariance that demands further investigation.

4. Semantically, it is clear that one should be able to internalize the judgement $\alpha : M \Longrightarrow_A N$ by a directed-Hom-type $\mathsf{Hom}_A(M, N)$ of transformations between $M, N : A$, as long as M is treated as a contravariant position. However, the introduction and elimination rules for this type require further study.
5. In addition to the admissibility Π, which is contravariant in its domain, there is a covariant Π-type corresponding the exponential object in the category of presheaves. We have given operationally plausible rules for this type, but have left the restrictions necessary to avoid impredicativity to future work. This covariant Π is analogous to parametric polymorphic quantification, and suggests a semantically motivated way of integrating higher-order abstract syntax into our approach.
6. Semantically, all of symmetric type theory sits inside of directed type theory, as every groupoid is a category. By exposing this structure in the syntax, we can ensure that directed type theory provides strictly greater flexibility than traditional type theories.
7. Thus far, we have considered only the two-dimensional case of directed type theory; higher-dimensional generalizations would admit more sophisticated constructions, which make use of transformations between transformations.

Once these issues are understood, the next step is implementation. Some issues are familiar from existing proof assistants. For example, we must outfit directed type theory with practical necessities like meta-variables and implicit argument reconstruction. Other issues are new: For example, in the current presentation, we have taken many equations for *resp* and map to hold definitionally. These include both the equations defining these constructs for each type and term, as well as associativity and unit laws. It remains to be seen whether this theory can be effectively implemented. One approach would be to isolate a decidable notion of definitional equality, and switch to a presentation that treats equality more like in intensional type theory, at which point we could prove a decidability result. Another would be to keep the theory in its current extensional form, and show that the equational theory, though undecidable in general, can be automated enough to be useful in practice.

To validate this work, we must show that directed type theory improves upon the examples that we have used for motivation here: For example, we hope to show that we can write down signatures for BL_0 and differential privacy and automatically derive the structural properties from them; and that we can provide a concrete syntax for derivability that is as convenient as Twelf. If we can succeed at this endeavor, and successfully integrate the representational power of LF with the computational power of Martin-Löf type theory, then we will significantly improve on current technology for programming with domain-specific logics, with broad applications to program verification and mechanized metatheory. On the one hand, our work permits computation to be used in the specification of logical systems, allowing more object-language concepts to be inherited from the meta-language. On the other, our work makes it easier to program with domain-specific logics and type systems, transforming more of programming language design into ordinary programming.

Bibliography

M. Abadi. Access control in a core calculus of dependency. In *Internatonal Conference on Functional Programming*, 2006.

M. Abadi. Variations in access control logic. In *International Conference on Deontic Logic in Computer Science*, pages 96–109. Springer-Verlag, 2008.

M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages*, pages 147–160. ACM Press, 1999.

M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretic Computer Science*, 342(1):3–27, 2005.

A. Abel. Miniagda: Integrating sized and dependent types. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality And Recursion in Interative Theorem Provers*, 2010.

T. Altenkirch. Extensional equality in intensional type theory. In *IEEE Symposium on Logic in Computer Science*, 1999.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, 2003.

T. Altenkirch and P. Morris. Indexed containers. In *IEEE Symposium on Logic in Computer Science*, pages 277–285, Washington, DC, USA, 2009. IEEE Computer Society.

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL 1999: Computer Science Logic*. LNCS, Springer-Verlag, 1999.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Programming Languages meets Program Verification Workshop*, 2007.

T. Altenkirch, J. Chapman, and T. Uustalu. Monads Need Not Be Endofunctors. In *Foundations of Software Science and Computational Structures*, volume 6014, chapter 21, pages 297–311. Springer Berlin Heidelberg, 2010.

S. Ambler, R. L. Crole, and A. Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *International Conference on Theorem Proving in Higher-Order Logics*, pages 13–30, London, UK, 2002. Springer-Verlag.

J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and*

Computation, 2(3):297–347, 1992a.

J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992b.

A. W. Appel and E. W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.

K. Avijit and R. Harper. A language for access control. Technical Report CMU-CS-07-140, Carnegie Mellon University, Computer Science Department, 2007.

K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2010.

A. Avron. Simple consequence relations. *Information and Computation*, 92(1):105–139, 1991.

S. Awodey and M. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 2009.

B. Aydemir and S. Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, 2010.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15, 2008.

J. C. Baez and M. Shulman. Lectures on n-categories and cohomology. Available from `http://arxiv.org/abs/math/0608420v2`, 2007.

E. Barzilay and S. Allen. Reflecting higher-order abstract syntax in nuprl. In *International Conference on Theorem Proving in Higher Order Logics*, 2002.

L. Bauer, S. Garriss, J. M. Mccune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey System. In *Proceedings of the 8th Information Security Conference*, pages 431–445. Springer Verlag LNCS, 2005.

G. Bellè, C. Jay, and E. Moggi. Functorial ML. In H. Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 1996.

F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Computer Science Logic*, 2008.

U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *IEEE Symposium on Logic in Computer Science*, 1991.

R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

E. Bishop. *Foundations of constructive analysis*. McGraw-Hill, New York, 1967.

J. Borgström, A. D. Gordon, and R. Pucella. Roles, Stacks, Histories: A Triple for Hoare. Technical Report MSR-TR-2009-97, Microsoft Research, 2009.

A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. Consistency of the theory of contexts. *Journal of Functional Programming*, 16(3):327–395, May 2006.

V. Capretta and A. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *Proceedings of TYPES 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2007.

L. Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished., 1990.

D. Chan. Constructive negation based on the completed database. In *International Conference on Logic Programming*, pages 111–125, 1988.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 3–14, New York, NY, USA, 2010. ACM.

A. Chaudhuri and D. Garg. PCAL: Language support for proof-carrying authorization systems. In *Proceedings of the 14th European Symposium on Research in Computer Security*, September 2009.

A. Chlipala. A certified type-preserving compiler from λ-calculus to assembly language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2008.

S. Chong, A. C. Myers, K. Vikram, and L. Zheng. Jif reference manual. Available from `http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`, February 2009.

T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control (extended abstract). In *Computer Security Foundations Workshop*, 2003.

A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2002.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.

Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.2*. INRIA, 2009. Available from `http://coq.inria.fr/`.

K. Crary. *Type Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

K. Crary. Typed compilation of inclusive subtyping. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

K. Crary. Explicit contexts in LF. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

N. G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

J. Despeyroux and P. Leleu. Primitive recursion for higher-order abstract syntax with dependant types. In *Workshop on Intuitionistic Modal Logics and Applications*, Tento, Italy, 1999.

J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

H. DeYoung and F. Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In *Workshop on Foundations of Computer Security*, 2009.

H. DeYoung, D. Garg, and F. Pfenning. An authorization logic with explicit time. In *IEEE Computer Security Foundations Symposium*, 2008.

I. Dinur and K. Nissim. Revealing information while preserving privacy. In *ACM Symposium on Principles of Database Systems*, 2003.

D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *International Joint Conference on Automated Reasoning*, pages 632–646. Springer, 2006.

D. Duggan and A. Compagnoni. Subtyping for object type constructors. In *Workshop On Foundations Of Object-Oriented Languages*, 1999.

C. Dwork and K. Nissim. Privacy-preserving datamining on vertically partitioned databases. In *International Cryptology Conference*, 2004.

C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theoretical Cryptography Conference*, 2006.

P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical Frameworks*, 1991.

P. Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science, pages 120–134. Springer, 1996.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 2000.

M. Fiore. Second-order and dependently-sorted abstract syntax. In *IEEE Symposium on Logic in Computer Science*, pages 57–68. IEEE Computer Society Press, 2008.

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *IEEE Symposium on Logic in Computer Science*, 1999.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Computer Science Logic*, 2007.

M. J. Gabbay. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 2010. To appear.

A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In *IEEE Symposium on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, June 2008.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *Types*

for Proofs and Programs, pages 210–225. Springer LNCS, 2004.

D. Garg. *Proof Theory for Authorization Logic and its Application to a Practical File System*. PhD thesis, Carnegie Mellon University, 2009a.

D. Garg. Proof search in an authorization logic. Technical Report CMU-CS-09-121, Computer Science Department, Carnegie Mellon University, April 2009b.

D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Computer Security Foundations Workshop*, pages 183–293, 2006.

D. Garg and F. Pfenning. PCFS: A proof-carrying file system. Technical Report CMU-CS-09-123, Carnegie Mellon University, 2009.

R. Garner. Two-dimensional models of type theory. *Mathematical. Structures in Computer Science*, 19(4):687–736, 2009.

P. Gaucher. A model category for the homotopy theory of concurrency. *Homology, Homotopy, and Applications*, 5(1):549–599, 2003.

G. Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39, 1935.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

J.-Y. Girard. On the unity of logic. *Annals of pure and applied logic*, 59(3):201–217, 1993.

J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

R. Godement. *Théorie des faisceaux*. Hermann, Paris, 1958.

L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 16 Sept. 1991.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1), 1993.

J. Hickey, A. Nogin, X. Yu, and A. Kopylov. Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection. In *ACM SIGPLAN International Conference on Functional Programming*, pages 172–183, New York, NY, USA, 2006. ACM.

D. Hilbert. Die grundlegung der elementaren zahlenlehre. *Mathematische Annalen*, 104:485–494, 1931.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

M. Hofmann. Semantical analysis of higher-order abstract syntax. In *IEEE Symposium on Logic in Computer Science*, 1999.

M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*. Oxford University Press, 1998.

P. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational*

Abstract Algebra. PhD thesis, Cornell University, 1995.

P. Jackson. The nuprl proof development system (version 4.2) reference manual and user's guide. Available from `http://www.nuprl.org/documents/Jackson/Nuprl4.2Manual.html`, 1996.

L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

K. Kariso. Integrating Agda and automated theorem proving techniques. Talk at Dependently Typed Programming Workshop, 2010.

S. Krishnamurthi. The CONTINUE server (or, How I administered PADL 2002 and 2003). In *International Symposium on Practical Aspects of Declarative Languages*, pages 2–16. Springer-Verlag, 2003.

R. Laemmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2003.

D. R. Licata and R. Harper. A universe of binding and computation. In *ACM SIGPLAN International Conference on Functional Programming*, 2009.

D. R. Licata and R. Harper. A monadic formalization of ML5. In *Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, July 2010.

D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *IEEE Symposium on Logic in Computer Science*, 2008.

P. L. Lumsdaine. Weak ω-categories from intensional type theory. In *International Conference on Typed Lambda Calculi and Applications*, 2009.

P. Martin-Löf. *Hauptsatz* for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, Amsterdam, 1971. North Holland.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium*. Elsevier, 1975.

P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

C. McBride. Outrageous but meaningful coincidences. In *ACM SIGPLAN Workshop on Generic Programming*, 2010.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 15 (1), 2004.

J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3-4):373–409, 1999.

F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

M. Miculan. Developing (meta)theory of λ-calculus in the theory of contexts. In *Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN)*, pages 65–81, 2001.

D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. Technical report, University of Pennsylvania, Department of Computer and Information Science, Aug. 1990.

D. Miller and A. F. Tiu. A proof theory for generic judgments: An extended abstract. In *IEEE Symposium on Logic in Computer Science*, pages 118–127, 2003.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

A. Momigliano, A. Martin, and A. Felty. Two-level hybrid: A system for reasoning using higher-order abstract syntax. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2007.

J. Morgenstern and D. R. Licata. Security-typed programming within dependently-typed programming. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

T. Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. Available as technical report CMU-CS-08-126.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, Portland, Oregon, 2006.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets (how to break anonymity of the netflix prize dataset). In *IEEE Symposium on Security and Privacy*, pages 111–125. IEEE Computer Society, May 2008.

B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's Type Theory, an Introduction*. Clarendon Press, 1990.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

A. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9), September 1982.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

F. Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, 1994.

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Conference on*

Programming Language Design and Implementation, pages 199–208, 1988.

F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction*, pages 202–206, 1999.

F. Pfenning and R. J. Simmons. Substructural operational semantics as ordered logic programming. In *IEEE Symposium on Logic In Computer Science*, pages 101–110, Los Alamitos, CA, USA, September 2009. IEEE Computer Society.

B. Pientka. Functional programming with higher-order abstract syntax and explicit substitutions. In *Programming Languages meets Program Verification*, 2006.

B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–382, 2008.

A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.

A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

A. M. Pitts. Alpha-structural recursion and induction. *Journal of the Association for Computing Machinery*, 53:459–506, 2006.

A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer-Verlag, 2009.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, 2008.

F. Pottier. Static name control for FreshML. In *IEEE Symposium on Logic in Computer Science*, 2007.

N. Pouillard and F. Pottier. A fresh look at programming with names and binders. In *ACM SIGPLAN International Conference on Functional Programming*, pages 217–228, September 2010.

J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, 2002.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in

Haskell. In *ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM, 2008.

P. Schroeder-Heister. Rules of definitional reflection. In R. L. Constable, editor, *IEEE Symposium on Logic in Computer Science*, pages 222–232, Montreal, Canada, June 1993. IEEE Computer Society Press.

C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

R. Seely. Modeling computations: a 2-categorical framework. In *IEEE Symposium on Logic in Computer Science*, pages 65–71, 1987.

P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. In *ACM SIGPLAN International Conference on Functional Programming*, 2007.

T. Sheard. Languages of the future. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, August 2003.

R. J. Simmons and B. Toninho. Logic programming in constructive provability logic. Available from `http://www.cs.cmu.edu/~rjsimmon/papers/modlog.pdf`, 2010.

A. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1993.

M. Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universitaet Erlangen-Nuernberg, 1998.

T. Streicher. *Semantics of type theory: correctness, completeness, and independence results*. Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 369–383. IEEE Computer Society, 2008.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming*, 2010.

B. van den Berg and R. Garner. Types are weak ω-groupoids. Available from `http://www.dpmms.cam.ac.uk/~rhgg2/Typesom/Typesom.html`, 2010.

J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *IEEE Computer Security Foundations Symposium*, June 2008.

R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

V. Voevodsky. The equivalence axiom and univalent models of type theory. Available from `http://www.math.ias.edu/~vladimir/Site3/home_files/`, 2010.

E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions On Computer Systems*, 12(1):3–32, 1994.

H. Xi. Applied type system (extended abstract). In *Post-workshop proceedings of TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer-Verlag, 2003.

H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3), 2008a. Special issue on "Classical Logic and Computation".

N. Zeilberger. Focusing and higher-order abstract syntax. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–369, 2008b.

N. Zeilberger. *The logical basis of evaluation order and pattern matching*. PhD thesis, Carnegie Mellon University, 2009.

N. Zeilberger. Polarity and the logic of delimited continuations. In *IEEE Symposium on Logic in Computer Science*, 2010.