TYPE-THEORETIC METHODOLOGY FOR PRACTICAL PROGRAMMING LANGUAGES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Karl Fredrick Crary

August 1998

TYPE-THEORETIC METHODOLOGY FOR PRACTICAL PROGRAMMING LANGUAGES

Karl Fredrick Crary, Ph.D.
Cornell University 1998

The significance of type theory to the theory of programming languages has long been recognized. Advances in programming languages have often derived from understanding that stems from type theory. However, these applications of type theory to practical programming languages have been indirect; the differences between practical languages and type theory have prevented direct connections between the two.

This dissertation presents systematic techniques directly relating practical programming languages to type theory. These techniques allow programming languages to be interpreted in the rich mathematical domain of type theory. Such interpretations lead to semantics that are at once denotational and operational, combining the advantages of each, and they also lay the foundation for formal verification of computer programs in type theory.

Previous type theories either have not provided adequate expressiveness to interpret practical languages, or have provided such expressiveness at the expense of essential features of the type theory. In particular, no previous type theory has supported a notion of partial functions (needed to interpret recursion in practical languages), and a notion of total functions and objects (needed to reason about data values), and an intrinsic notion of equality (needed for most interesting results). This dissertation presents the first type theory incorporating all three, and discusses issues arising in the design of that type theory.

This type theory is used as the target of a type-theoretic semantics for a expressive programming calculus. This calculus may serve as an internal language for a variety of functional programming languages. The semantics is stated as a syntax-directed embedding of the programming calculus into type theory.

A critical point arising in both the type theory and the type-theoretic semantics is the issue of admissibility. Admissibility governs what types it is legal to form recursive functions over. To build a useful type theory for partial functions it is necessary to have a wide class of admissible types. In particular, it is necessary for all the types arising in the type-theoretic semantics to be admissible. In this dissertation I present a class of admissible types that is considerably wider than any previously known class.

Biographical Sketch

Karl Crary was born in Madison, Wisconsin in 1971. In 1977 he moved with his family to Seattle, Washington where he grew up. He started college at Carnegie Mellon University in 1989, and earned bachelor's degrees in Computer Science and Economics in 1993. He then began graduate school at Cornell University where he remained until 1998.

Acknowledgements

I wish to thank my parents, Fred and Betsy, for their constant love and support throughout my childhood and adulthood. More than anyone else, they have shaped the way I see and think about the world. I want to thank Ross Willard for introducing me to logic and modern mathematics, and John Reynolds for introducing me to the world of type theory. I also am very grateful to Robert Harper for serving as my undergraduate thesis advisor, for teaching me so much, and for all his encouragement while I was his student and continuing even today.

Particular thanks go to Robert Constable, my graduate advisor, and to Greg Morrisett. The least they did was to teach me most of the material on which this dissertation, and the rest of my work, is based. More importantly they were a constant source of ideas, sound advice and encouragement. I also wish to thanks Dexter Kozen for all his help and influence.

I also wish to thank all the many teachers, family and friends who helped me get where I am today, and who made it pleasant getting here. Finally, I want to thank God for creating the world, and for showing me the life beyond.

Table of Contents

List of Figures

Chapter 1

Introduction

The distance between theory and practice is shorter in theory than in practice.
—Unknown

The development of practical programming languages has lagged far behind the state-of-the-art of theoretical programming language study. Examples abound of programming language techniques and methods that have been well known and understood for years, but still have yet to appear in any practical programming system. This can be frustrating to the programming language theoretician who also writes code, and that frustration is not diminished by the knowledge that the gap between theory and practice is common to many, if not most, fields of study.

The main reason that the gap between theory and practice is often so wide is the issue of elegance. Programming languages can admit extremely elegant theoretical study. This elegance is partly responsible for the rapid progress of results in theoretical programming languages, and it is partly responsible for the attractiveness of programming languages as a field of study. However, such elegance is usually achieved by abstracting away from the details of real languages—details that are essential for real languages to be usable.

In my thesis research, I set out to design and implement a practical programming language to incorporate unutilized developments in programming language theory. The result was a dialect of ML that came to be called KML. Not surprisingly, I was soon grappling with the elegance gap between theory and practice. To manage the complexity of a real system, I allowed type theory to drive the design. That is, I adopted the strategy of choosing, at every design point, an option that was easily represented and justified in an underlying foundation of type theory.

This strategy proved remarkably effective, and resulted in a language design that was versatile and powerful but also relatively simple. However, allowing type theory to drive the design of KML paid another dividend, which had wider applications than to the KML language alone. The type-theoretic design of KML led me to explore not only how type theory can motivate language design informally, but also to explore how (and why) to draw formal connections between practical programming languages and foundational type theory.

This dissertation focuses on the latter aspect of my thesis research, formal connections between practical programming languages and foundational type theory. In this thesis we will see various specific applications of such connections, but from a broader standpoint the aim of the work I discuss here is to draw theory and practice closer together. The constructs that make up type theory are powerful and uniform, making it possible to reason about issues facing real languages without sacrificing elegance.

1.1 The KML Programming Language

A complete discussion of the KML programming language is beyond the scope of this dissertation. I give here a brief and informal introduction to KML and its contributions.

1.1.1 Features

The main objective of KML was first to incorporate into a practical dialect of the ML programming language a number of useful language features not present in preceding dialects of ML while building the entire language on a solid theoretical foundation. KML is structured to greatly resemble Standard ML; for the most part, Standard ML is a strict subset of KML. The differences between KML and Standard ML lie in the way KML is specified, and in the additional language features supported by KML: object-oriented programming, subtyping, a more expressive module system and first-class polymorphism. These features are chosen because they can assist in a real and practical way with the task of programming.

A KML compiler is implemented that consists of a KML-specific front end, which compiles KML to an intermediate language based on the polymorphic lambda calculus, and a back end that compiles that intermediate language to Typed Assembly Language [77, 76].

At the time of this writing, the design of KML is substantially complete but is still changing in a number of areas. Therefore, the reader should take the following descriptions of KML's features as tentative.

Objects Object-oriented programming is supported in KML through a primitive object calculus [31] that is substantially similar to that of Abadi and Cardelli [1, 2]. Objects are seen as collections of method functions, each of which takes the entire object as an argument and computes a result [58]. A method's results may be a new object of the same type; such results are usually produced by updating one of the fields of the object with a new method implementation.

In the present design, primitive objects are the extent of the support for object-oriented programming. From primitive objects it is possible to code more complicated artifices like classes and inheritance. I plan built-in syntactic support for classes and inheritance in a future version of the design.

Subtyping The KML type system completely integrates support for subtyping. Subtyping arises by dropping methods from objects and fields from records, and also from stated definitions. Definitional subtyping happens in bounded quantification, where code is written to be polymorphic over all types that are subtypes of a given type, and in abstract subtyping, where a type in a module is specified to be a subtype of some other type. KML allows unrestricted subsumption, that is, a member of a subtype may always be used in place of a member of the supertype without any mediating syntax such as a coercion.

Expressive Modules The KML module system builds on the module system of Standard ML with two main enhancements. The first is higher-order functors. While Standard ML functors may take and return only structures, the KML module system allows functors to take other functors as arguments and return functors as results. Thus, the KML module system is fully featured typed lambda calculus.

The second enhancement is to allow type definitions in signatures. Allowing type definitions in signatures, combined with a type system that leaks no type information in the absence of such definitions, allows so-called "translucent" modules, modules that leak exactly the desired amount of type information. Translucent modules allow the greatest amount of abstraction possible in a system, while still permitting enough leakage to link modules together.

First-Class Polymorphism Most dialects of ML, including Standard ML, restrict polymorphic terms so that they cannot be placed into data structures, passed to functions or returned by functions. (Formally, this restriction stems from a prenex restriction on quantified types: quantifiers must appear all the way out, which means that quantified types cannot make up a component of any larger type.) Thus, polymorphic values are not first class; they must always be instantiated before they may be used. In contrast, polymorphic values in KML are first class; they may be arguments or results of functions and may be stored in data structures.

KML also permits polymorphic recursion [78], where a polymorphic function calls itself recursively with a different type argument. Polymorphic recursion is necessary to make good use of first-class polymorphism (otherwise all type arguments are statically predetermined), but it is also useful in other contexts [79].

1.1.2 Comparison

During the time I was developing KML, Xavier Leroy and his colleagues released the Objective CAML system [64]. Objective CAML shares many of the same goals as KML, in particular the goals of providing support for object-oriented programming and a more expressive module system. However, Objective CAML does not support first-class polymorphism and possesses no formal semantics or definition.

The greatest similarity between KML and Objective CAML lies in the module systems, which provide almost identical expressiveness, despite various technical differences. A greater contrast exists in the object system. The object system of KML is derived from the object calculus of Crary [31], which derives its lineage from the object systems of Abadi and Cardelli [1, 2] and their predecessors. In contrast, the object system of Objective CAML is based on the object system of Rémy and Vouillon [85], which derives its lineage from the record extension calculi of Rémy [84].

The greatest practical difference between the object systems of KML and Objective CAML lies in support for subtyping. Unlike KML, Objective CAML provides no automatic subsumption; objects must be coerced to supertypes by an explicit coercion. However, by abandoning subtyping with subsumption, Objective CAML is able to enjoy feasible type inference, which is prohibitively difficult for KML. Moreover, in many cases the practical benefits of subtyping are obtained in Objective CAML by quantification over row variables that extend object types to unspecified supertypes (details appear in Rémy and Vouillon [85] and Leroy [64]).

1.1.3 The Role of Type Theory in the Design

In order to produce an elegant and coherent design for KML, I allowed type-theoretic considerations to drive the design. In order to convey the flavor of this interaction, I briefly describe two design points that were informed by type theory.

In a module calculus, one important issue to settle is the equational theory: under what circumstances are modules and components of modules considered equal to others. The definition of Standard ML [71] employed a system of unique stamps to dictate when two modules were considered equal. This system becomes even more involved when extended to higher-order modules [66]. In KML, I simplified the equational theory by eliminating equality of modules in favor of equality of the types that make up those modules (a similar choice was made for

Standard ML (Revised) [72]), and then ruling that types are equal exactly when equivalence may be proven from stated type definitions in a logic of types. (Leroy made a similar decision in the design of the Objective CAML module system.)

Another key design point was what form support for object-oriented programming should take. Rather than begin with a source level system of objects or classes, I focused on the type-theoretic interpretation of objects. With an theory of objects in hand, I worked backwards to the KML object calculus. The result was an object system with comparable power to that of Abadi and Cardelli [1, 2], but with a simpler theory and a more efficient implementation. Details appear in Crary [31].

1.2 Principals

1.2.1 Type Theory

One may view foundational type theory as being, at its core, a typed functional programming language with a very rich type system. That type system is capable of expressing complex relationships between objects; sufficiently complex, that it may encode (almost) any sentence in logic and mathematics. In this way, type theory may serve as a foundation for mathematics and computation.

To some degree, this view reflects the bias of a computer scientist. Type theory has been used as a foundation for mathematics since before the advent of modern computers; for instance, Whitehead and Russell's influential 1910 treatise *Principia Mathematica* [96] was a type theory in some ways similar to the type theory of this dissertation. Nevertheless, type theory has been particularly successful as a foundation for computer science. This is largely because type theory deals elegantly with structured data and computation (the "stuff" of computer science), which must be painstakingly constructed in most other foundational frameworks.

The particular type theory I use in this dissertation is a variant of the Nuprl type theory [19]. Nuprl traces its lineage back to Martin-Löf's Intuitionistic Type Theory [67], which was first intended to clarify the syntax and semantics of intuitionistic mathematics. The Nuprl type theory adapted Intuitionistic Type Theory as a foundation for computer programming by simplifying its structure and adding a number of new type constructors. Although my results are stated in the context of the Nuprl type theory, my intention is that the methodology behind those results be applicable to type theories in general.

1.2.2 Lambda-K

In order to manage the complexities of a real programming language, KML is defined in terms of a smaller internal language called λ^K. Lambda-K provides all the expressiveness of the full KML language, but is considerably simpler and more elegant, because many of the complex constructs of KML are broken up by the definition into simpler constructs in λ^K.

The main thrust of this dissertation, then, is an exploration of how to draw formal connections between the internal language λ^K and the Nuprl type theory. These formal connections then apply back to the high-level language KML by way of its definition. However, I wish to view the KML particulars as only a case study in the general methodology I explore. Similar results can be drawn for other languages; for example, Harper and Stone [48] give an interpretation of Standard ML in terms of an internal language very much like λ^K. Thus, with some minor modifications, my results would apply to Standard ML as well. Accordingly, I do not present the formal definition of KML in λ^K in this dissertation; to do so would unduly focus on the particulars and distract from the broader methodology.

The Nuprl type theory is predicative, so in order to draw formal connections to Nuprl from λ^K, it is necessary for λ^K to be predicative as well. However, the λ^K used in defining KML is not predicative. Consequently, in this dissertation I use a predicative variant of λ^K that drops a number of features. Recursive types, stateful computation and objects all depend on impredicativity in some way, and are regrettably omitted. Restoring these is an important avenue for future research, and I discuss some possible ways it might be done in Section 3.5.

1.3 Overview

After this introductory chapter, the technical material commences in Chapter 2 with an exposition KML's internal language λ^K. In keeping with the view of λ^K as a case study, that chapter is brief. The primary original contribution of λ^K is a module calculus that combines contributions from the translucent sums calculi of Harper and Lillibridge [44] and Leroy [62], the phase-distinction calculus of Harper, Mitchell and Moggi [46], and the applicative functors calculus of Leroy [63].

Chapter 3 begins with an informal description of the Nuprl type theory, and then continues with an embedding of λ^K into Nuprl, the first of the three main technical contributions of this dissertation. That embedding provides a type-theoretic semantics of λ^K, and thereby draws a sharp formal connection between KML and type theory. I also discuss at length the applications and some of the consequences of the semantics.

In Chapter 4 I discuss in detail my variant of the Nuprl type theory, the second main technical contribution of this dissertation. This type theory is the first constructive type theory to include an intrinsic notion of equality and also to allow reasoning about both partial and total functions. All three notions are necessary to give an adequate account of practical programming languages; partiality is clearly necessary for Turing-complete languages, and equality and totality are necessary to say anything interesting about such languages once interpreted. I also discuss practical issues that arise when using a type theory incorporating partiality. Finally, I give a rigorous semantics to the type theory and show its consistency.

Central to the type theory is a fixpoint rule for typing recursive functions. However, the fixpoint rule is not valid for recursive functions on every type. Types for which the fixpoint rule is valid are known as *admissible*. In Chapter 5 I give general techniques for showing types to be admissible, the third main technical contribution of this dissertation. Before this work, type theories incorporating partiality were handicapped by a dearth of types known to be admissible. A critical step in the method is a elegant least upper bound theorem for computational approximation with applications beyond the admissibility results to which I apply it.

Concluding remarks appear in Chapter 6. Following my concluding remarks, the dissertation closes with three appendices. Appendix A gives the complete typing rules for λ^K. Appendix B gives the complete inference rules for Nuprl proofs. In the interest of readability throughout the dissertation, difficult proofs are removed to Appendix C.

Chapter 2

Lambda-K

The KML programming language provides a very expressive language for practical programming. However, experience with Standard ML has shown that although direct formal definitions of practical programming languages may be written [71, 72], those definitions are too unwieldy to be very useful except as a language specification for compiler implementers and are particularly impractical for theoretical study.[1] Simple languages are much more practical for theoretical study, and are useful in implementations as well: The first step of a compiler is typically to elaborate the external language into a simpler intermediate language.

In order to use a simpler and more manageable language than full KML, in this thesis I will be using a simple typed calculus called λ^K. A framework that proved to be workable was to augment the higher-order polymorphic lambda calculus, F_ω [35, 36], with power [17, 18] and singleton [41] kinds, dependent function and record kinds, and strong sums. This design is discussed at length in this chapter.

The syntax of λ^K is defined by the rules given in Figure 2.1 and consists of five syntactic classes. The class of *terms* contains the basic constructs of the polymorphic lambda calculus with records. The class of *type constructors* (which I will usually refer to as "constructors") contains the types of well-formed terms and a lambda calculus (with records) built over them for computing types. (Typical type constructors are actual types, such as integers, or (first-order) type operators, such as lists.) The class of *kinds* then contains the "types" of type constructors. The two remaining classes, *modules* and *signatures*, contain the terms and types of λ^K's module system. The static semantics of λ^K also uses *contexts*, which assign kinds, types and signatures to constructor variables (ranged over by α), term variables (ranged over by x) and module variables (ranged over by s). Figure 2.2 contains several convenient abbreviations for use with λ^K.

Throughout this thesis, in any calculi with binding structure (such as λ^K), I consider two expressions to be identical if they differ only by alpha-variation. When two expressions e_1 and e_2 are alpha-equal I write $e_1 \equiv e_2$. Also, the simultaneous capture-avoiding substitution of expressions $e_1, \ldots, e_n$ for variables $x_1, \ldots, x_n$ in e will be denoted by $e[e_1 \cdots e_n / x_1 \cdots x_n]$.

[1] The Definition of Standard ML [71] runs to 65 pages and consists of 196 rules (plus many additional implied rules for propagating exception packets) and considerable supporting machinery. The revised Definition [72] is simpler, but still runs to 54 pages and 189 rules.

kinds	κ	::=	$Type_i \mid \Pi\alpha{:}\kappa_1.\kappa_2 \mid \{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\} \mid \mathcal{P}_i(c) \mid \mathcal{S}_i(c)$
constructors	c	::=	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1[c_2] \mid \{\ell_1 = c_1, \ldots, \ell_n = c_n\} \mid \pi_\ell(c) \mid c_1 \rightarrow c_2 \mid c_1 \Rightarrow c_2 \mid$ $\forall\alpha{:}\kappa.c \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \mid \langle \ell_1 : c_1, \ldots, \ell_n : c_n\rangle \mid ext(m)$
terms	e	::=	$x \mid \lambda x{:}c.e \mid e_1 e_2 \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid \pi_\ell(e) \mid inj_\ell(e) \mid$ $case(e, \ell_1 \triangleright x_1.\, e_1, \ldots, \ell_n \triangleright x_n.\, e_n) \mid fix_c(e) \mid ext(m)$
signatures	σ	::=	$\langle\kappa\rangle \mid \langle\!\langle c\rangle\!\rangle \mid \Pi s{:}\sigma_1.\sigma_2 \mid \{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}$
modules	m	::=	$s \mid \langle c\rangle \mid \langle\!\langle e\rangle\!\rangle \mid \lambda s{:}\sigma.m \mid m_1 m_2 \mid \{\ell_1 = m_1, \ldots, \ell_n = m_n\} \mid \pi_\ell(m) \mid m : \sigma$
contexts	Γ	::=	$\bullet \mid \Gamma[\alpha : \kappa] \mid \Gamma[x : c] \mid \Gamma[s : \sigma]$

Figure 2.1: Lambda-K Syntax

$$
\begin{aligned}
\tau_1 \times \cdots \times \tau_n &\stackrel{\text{def}}{=} \{l1 : \tau_1, \ldots, ln : \tau_n\} \\
\langle e_1, \ldots, e_n\rangle &\stackrel{\text{def}}{=} \{l1 = e_1, \ldots, ln = e_n\} \\
\pi_i(e) &\stackrel{\text{def}}{=} \pi_{li}(e) \\
\tau_1 + \cdots + \tau_n &\stackrel{\text{def}}{=} \langle l1 : \tau_1, \ldots, ln : \tau_n\rangle \\
inj_i(e) &\stackrel{\text{def}}{=} inj_{li}(e) \\
case(e, x_1.e_1, \ldots, x_n.e_n) &\stackrel{\text{def}}{=} case(e, l1 \triangleright x_1.\, e_1, \ldots, ln \triangleright x_n.\, e_n)
\end{aligned}
$$

(where $l1, l2, \ldots$ are distinguished labels)

Figure 2.2: Lambda-K Shorthand

2.1 The Core Calculus

The type system of λ^K includes functions and records at the term and constructor levels, and polymorphic functions at the term level, using the standard notations. Evaluation is intended to be call-by-value. Records of terms, record types, and records of constructors that differ only in field order are considered identical. The type of functions from τ_1 to τ_2 is denoted by $\tau_1 \rightarrow \tau_2$; functions that return without computational effects are considered *pure* functions and may be given the pure function type $\tau_1 \Rightarrow \tau_2$ (for appropriate τ_1 and τ_2). The type system also includes the labelled disjoint union type, denoted by $\langle \ell_1 : c_1, \ldots, \ell_n : c_n\rangle$, members of which are formed by $inj_\ell(e)$ and eliminated by $case(e, \ell_1 \triangleright x_1.\, e_1, \ldots, \ell_n \triangleright x_n.\, e_n)$. As with records, disjoint union types or case expressions that differ only in field order are considered identical. Recursive functions are supported in the standard way by a *fix* operator. The remaining construct, $ext(m)$, deals with the module calculus and is discussed in Section 2.2.

A subtyping relation is defined over the types of λ^K. Intuitively, τ_1 is a subtype of τ_2 (denoted $\tau_1 \preceq \tau_2$) when every term belonging to τ_1 also belongs to τ_2. As usual, functions are contravariant on the left and covariant on the right; records are covariant in the field types and subtypes are produced by adding fields. Pure function types are subtypes of the corresponding function types.

Kinds The kind level is more novel. Ignoring the i annotations, *Type* contains well-formed types. The power kind, denoted by $\mathcal{P}(\tau)$ for any type τ, contains well-formed types that are subtypes of the given type τ. The singleton kind, denoted by $\mathcal{S}(\tau)$ for any type τ, contains well-formed types that are equivalent to the given type τ. The level annotations i, which are positive integers, restrict the memberships of these kinds. The level of a kind or type is defined to be the highest level annotation appearing within it. The kinds $Type_i$, $\mathcal{P}_i(\tau)$ and $\mathcal{S}_i(\tau)$ contain

$$\begin{array}{rcl}\mathcal{R}(c : \mathit{Type}_i) & \stackrel{\text{def}}{=} & \mathcal{R}_i(c) \\ \mathcal{R}(c : \kappa_1 \rightarrow \kappa_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}\kappa_1.\mathcal{R}(c[\alpha] : \kappa_2) \\ \mathcal{R}(c : \{\ell_i : \kappa_i{}^{[i=1\ldots n]}\}) & \stackrel{\text{def}}{=} & \{\ell_i : \mathcal{R}(\pi_{\ell_i}(c) : \kappa_i)^{[i=1\ldots n]}\}\end{array}$$

($\mathcal{R}$ represents either $\mathcal{P}$ or $\mathcal{S}$)

Figure 2.3: Higher Order Power and Singleton Kinds

only types with levels less than i. Thus, types belonging to Type_1 cannot contain quantify over any (nontrivial) kinds, and types belonging to Type_2 can only quantify over level-1 kinds. For $\mathcal{P}_i(c)$ or $\mathcal{S}_i(c)$ the level of c must be less than i.

This level annotation mechanism makes λ^K predicative, which is necessary to perform Chapter 3's embedding into the predicative type theory of Nuprl. Since these annotations are tedious to provide, I discuss some alternatives in Section 3.5. In a practical system based on this annotation mechanism, the system would have to infer level annotations for the user [47].

The kind of functions mapping constructors of kind κ_1 to constructors of kind κ_2 may be denoted $\kappa_1 \rightarrow \kappa_2$, but since constructors may appear within kinds, it is desirable to allow a more precise characterization of such functions: The kind $\Pi\alpha{:}\kappa_1.\kappa_2$ includes all functions mapping κ_1 constructors to κ_2 constructors where α stands for the function's argument and may appear free in κ_2. Such kinds are referred to as *dependent* function kinds since the result kind depends on the argument. For example, a function with kind $\Pi\alpha{:}\mathit{Type}_i.\mathcal{P}_i(\alpha)$ when applied to any type τ returns a constructor of kind $\mathcal{P}_i(\tau)$, that is, a subtype of τ. The *independent* function kind $\kappa_1 \rightarrow \kappa_2$ is shorthand for $\Pi\alpha{:}\kappa_1.\kappa_2$ when α does not appear free in κ_2.

Records of type constructors may also be given dependent kinds. The kind $\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}$ contains all records $\{\ell_1 = c_1, \ldots, \ell_n = c_n\}$ such that each c_i has the corresponding kind κ_i with the values of preceding fields substituted for the free variables. That is, c_i must have kind $\kappa_i[c_1 \cdots c_{i-1}/\alpha_1 \cdots \alpha_{i-1}]$. The external labels (ranged over by ℓ) must be kept distinct from the internal binding occurrences because variables must alpha-vary but labels must not.[2] This is discussed in greater length in Harper and Lillibridge [44]. As a shorthand, I will often omit $\triangleright\,\alpha$ (pronounced "as alpha") from a dependent record kind when α does not appear free in the following kinds; an *independent* record kind is one that contains no internal binding occurrences at all. I consider two dependent record kinds to be identical when they differ only in field order, so long as no field is reordered to appear after a field that depends upon it.

Although power and singleton kinds are made using only types (not higher-order constructors), dependent kinds may be used to build higher-order power and singleton kinds. For example, the kind $\Pi\alpha{:}\mathit{Type}_i.\mathcal{P}_i(\mathit{list}[\alpha])$ includes type constructors that are pointwise subtypes of *list*. Higher-order power and singleton kinds may be generally defined as in Figure 2.3.

Just as a subtyping relation was defined over the types, a *subkinding* relation is defined over the kinds. This relation is denoted by $\kappa_1 \preceq \kappa_2$ when κ_1 is a subkind of κ_2. Function and record kinds obey similar rules to the subtyping rules. Additional subkinding relationships result from level annotations, $\mathit{Type}_i \preceq \mathit{Type}_{i+1}$ (and likewise for singleton and power kinds), and from power and singleton kinds: for any type τ, $\mathcal{S}_i(\tau) \preceq \mathcal{P}_i(\tau) \preceq \mathit{Type}_i$, and also $\mathcal{P}_i(\tau_1) \preceq \mathcal{P}_i(\tau_2)$ when $\tau_1 \preceq \tau_2$.

[2]These two names may be burdensome for a programmer to supply, so, as a practical matter, in KML it is permitted to write one name for use as both the internal and the external name; this poses no problems so long as it is possible to use separate names when they are needed [44].

```
sig
   tycon foo : type
   module S1 :
      sig
         tycon bar : type
         tycon baz = foo -> bar
         val gnurf : baz
      end
   val blap : S1.bar
end
```

$$\Longrightarrow \quad \begin{array}{l} \{\ \texttt{foo} \triangleright s_{\mathrm{foo}} : \langle \mathit{Type} \rangle, \\ \quad \texttt{S1} \triangleright s_{\mathrm{S1}} : \\ \qquad \{\ \texttt{bar} \triangleright s_{\mathrm{bar}} : \langle \mathit{Type} \rangle, \\ \qquad\quad \texttt{baz} \triangleright s_{\mathrm{baz}} : \langle \mathcal{S}(\mathit{ext}(s_{\mathrm{foo}}) \rightarrow \mathit{ext}(s_{\mathrm{bar}})) \rangle, \\ \qquad\quad \texttt{gnurf} : \langle\!\langle \mathit{ext}(s_{\mathrm{baz}}) \rangle\!\rangle\ \}, \\ \quad \texttt{blap} : \langle\!\langle \mathit{ext}(\pi_{\texttt{bar}}(s_{\mathrm{S1}})) \rangle\!\rangle\ \} \end{array}$$

```
struct
   tycon foo = int -> int
   module S1 =
      struct
         tycon bar = int
         tycon baz =
            (int -> int) -> int
         val gnurf =
            fn f : int -> int => f 0
      end
   val blap = 12
end
```

$$\Longrightarrow \quad \begin{array}{l} \{\ \texttt{foo} = \langle \mathit{int} \rightarrow \mathit{int} \rangle, \\ \quad \texttt{S1} = \\ \qquad \{\ \texttt{bar} = \langle \mathit{int} \rangle, \\ \qquad\quad \texttt{baz} = \langle (\mathit{int} \rightarrow \mathit{int}) \rightarrow \mathit{int} \rangle, \\ \qquad\quad \texttt{gnurf} = \langle\!\langle \lambda f{:}(\mathit{int} \rightarrow \mathit{int}).\, f\, 0 \rangle\!\rangle\ \}, \\ \quad \texttt{blap} = \langle\!\langle 12 \rangle\!\rangle\ \} \end{array}$$

Figure 2.4: A Typical Module Encoding

2.2 The Module Calculus

Lambda-K modules form their own syntactic class, and their types are called signatures, which also form their own syntactic class. A primitive module is either a single constructor $\langle c \rangle$ or a single term $\langle\!\langle e \rangle\!\rangle$. If c has kind κ, then $\langle c \rangle$ has signature $\langle \kappa \rangle$. Likewise, if e has type τ, then $\langle\!\langle e \rangle\!\rangle$ has signature $\langle\!\langle \tau \rangle\!\rangle$.

Modules are also formed using lambda-abstractions (for functors) and records (for structures). Such modules may be given dependent function signatures and dependent record signatures that are precisely analogous to the dependent function and dependent record kind discussed above. Data abstraction is achieved by forgetting type information using the construct $m : \sigma$, which coerces the module m to have the signature σ (if that signature is valid for m); any type information about m not reflected in σ is forgotten. As with the subtyping and subkinding relations over the types and kinds, the signatures have a *subsignature* relation, which obeys rules similar to the rules governing subtyping and subkinding.

Modules are used within the core calculus by means of the extraction construct: $\mathit{ext}(m)$ for module m. If m has signature $\langle \kappa \rangle$, then the constructor $\mathit{ext}(m)$ has kind κ. Likewise, if m has signature $\langle\!\langle \tau \rangle\!\rangle$ then the term $\mathit{ext}(m)$ has type τ. Exactly what modules are legal to extract from is an issue that is discussed in Section 2.4.1.

With the above constructs, it is easy to encode high-level KML modules. A KML structure is constructed by building a record out of its contents with every constructor field encased in $\langle \cdot \rangle$ and every term field encased in $\langle\!\langle \cdot \rangle\!\rangle$. Figure 2.4 shows the encoding of a typical KML module with a substructure into λ^K. Note the use of an internal and an external name in the encoding of the `blap` field of the signature. Functors and their signatures are encoded using λ and Π in the obvious manner.

Since modules form their own syntactic class and are not reflected directly into the core calculus, λ^K modules are second-class. This decision is based on issues related to the phase distinction (Section 2.4.1). To permit first-class modules would require considerably different approaches to these issues that would reduce the expressiveness of the calculus, as we will see shortly. Fortunately, the first-class module restriction does not seem to be an onerous one: the most commonly cited use for a first-class module, the ability to select at run-time an efficient implementation, can be achieved using existential types [73] or objects.

2.3 Static Semantics

The static semantics of λ^K appears in Appendix A. Nine judgements are used; six for the core calculus and three for the module calculus:

1. The kind equality judgement $\Gamma \vdash_K \kappa_1 = \kappa_2$ indicates that κ_1 and κ_2 are equal (in context Γ).
2. The subkinding judgement $\Gamma \vdash_K \kappa_1 \preceq \kappa_2$ indicates that every constructor in κ_1 is in κ_2, as discussed previously.
3. The constructor equality judgement $\Gamma \vdash_K c_1 = c_2 : \kappa$ indicates that c_1 and c_2 are equal as members of kind κ.
4. The subtyping judgement $\Gamma \vdash_K c_1 \preceq c_2$ indicates that every term in c_1 is in c_2.
5. The typing judgement $\Gamma \vdash_K e : c$ indicates that the term e has the type c.
6. The valuability judgement $\Gamma \vdash_K e \downarrow c$ indicates that the term e has the type c and that its computation terminates.
7. The subsignature judgement $\Gamma \vdash_K \sigma_1 \preceq \sigma_2$ indicates that every module in σ_1 is in σ_2.
8. The module signature judgement $\Gamma \vdash_K m : \sigma$ indicates that the module m has signature σ.
9. The module valuability judgement $\Gamma \vdash_K m \downarrow \sigma$ indicates that the module m has signature σ and that its computation terminates.

Four additional judgements are derived from these nine. A kind is considered well-formed when it is equal to itself, and similarly a constructor belongs to a kind exactly when it is equal to itself in that kind. A type or signature is well-formed when it is a subtype or subsignature of itself.

$$\begin{array}{rcl}
\Gamma \vdash_K \kappa \text{ kind} & \stackrel{\text{def}}{=} & \Gamma \vdash_K \kappa = \kappa \\
\Gamma \vdash_K c : \kappa & \stackrel{\text{def}}{=} & \Gamma \vdash_K c = c : \kappa \\
\Gamma \vdash_K c : \text{type} & \stackrel{\text{def}}{=} & \Gamma \vdash_K c \preceq c \\
\Gamma \vdash_K \sigma \text{ sig} & \stackrel{\text{def}}{=} & \Gamma \vdash_K \sigma \preceq \sigma
\end{array}$$

2.4 Design Issues

2.4.1 The Phase Distinction

An important decision to be made in the design of any module calculus is how to account for the phase distinction between *compile-time* and *run-time* expressions [16, 46, 44]. In the absence of the module calculus, it is easy to say that the evaluation of type constructors is a compile-time activity and the evaluation of terms is a run-time activity. However, the module system presents a quandary since modules may contain terms and type constructors may contain modules (through the extraction construct). Constructor evaluation cannot be a run-time activity, unless we are willing to sacrifice type checking at compile time, but term evaluation clearly cannot be a compile-time activity.

The central problem is how to structure the module system and its interactions with the core calculus so that compile-time expressions (kinds and constructors) do not contain possibly divergent or effectful expressions. I discuss here two solutions to the problem: the phase-splitting approach proposed by Harper, Mitchell and Moggi [46] and the value restriction approach proposed by Harper and Lillibridge [44] and Leroy [62]. Lambda-K makes use of the phase-splitting approach.

In the phase-splitting approach, modules (including functors) are considered to consist of two components: a compile-time component and a run-time component. The compile-time component deals only with the constructor portions of modules while the run-time component deals also with the term portions. Then extraction at the constructor level depends only on the constructor portion of a module; any problematic behavior of the module's term portion is irrelevant. This approach depends on designing the module calculus so that it is impossible to write a module where a constructor field depends on any run-time computation; otherwise constructor fields are run-time computations themselves and cannot be phase split into the compile-time component.

In a system with first class modules, it is easy to construct a module that depends on the result of a term expression, which rules out the phase-splitting approach. In their translucent sum calculus [44], which has first class modules, Harper and Lillibridge enforce the phase distinction more directly: Syntactic values have no interesting run-time behavior and may therefore safely be treated as compile-time expressions. Extraction at the constructor level is restricted to only valuable expression; arbitrary module expressions are not legal for extraction.

The phase-splitting approach also has some advantages that counterbalance the impossibility of first-class modules: First, phase-splitting provides a convenient mechanism for implementing the module system [46]. Second, phase-splitting allows greater flexibility in references from the core language into the module language. For example, phase-splitting allows extraction from functor applications, which is forbidden by the value restriction approach. This can be quite useful in practice; for example, it allows the expression of transparent signatures for functors [63], that is, signatures that completely specify the (type) behavior of the functor. For example, suppose s has signature $\Pi s'{:}\sigma.\langle \mathit{Type}\rangle$. Then the alternative signature $\Pi s'{:}\sigma.\langle \mathit{ext}(s\,s')\rangle$ fully specifies the behavior of s, but this signature is valid only if it is permissible to extract from functor applications. Leroy [63] discusses this issue at length.

2.4.2 Top and Bottom Types

Unlike many subtyping calculi, λ^K does not have top and bottom types. This is not because they are problematic to add (they are not), but because a practical language is better off without them. The main use for top and bottom is usually to ensure that there exist meets and joins for any two types, but it is really only important to ensure that joins and meets exist for types that have an upper or lower bound. In cases where two types have no upper (or lower) bound, it is preferable to generate an error than to allow an artificial top (or bottom) bound. For example, a program fragment `if b then 12 else true` is almost certainly a mistake. Similarly, a function that makes incompatible demands on its argument could be given a bottom domain type, but again such a useless function is certainly a mistake. It is better the notify the programmer of the mistake immediately than to allow the trivial type and postpone the error message, delivering it somewhere else instead of at the location where the mistake was made.

The drawback of this design decision relates to expressions with side-effects. When an expression is evaluated for side-effects, it is not uncommon to ignore the result value. In λ^K this must be done explicitly, as in the code `if b then f x; () else g x; ()`, where `f` and `g` have incompatible return types. If the language included a top type, the programmer could write the marginally simpler `if b then f x else g x`. However, even in very impure code, it is not likely that allowing top and bottom would save effort for the programmer in this manner nearly as often as disallowing them would save effort by catching mistakes. Additionally, writing code to ignore result values explicitly provides useful documentation.

Chapter 3

Type-Theoretic Semantics

Type theory has become a popular framework for formal reasoning in computer science [22, 67, 37] and has formed the basis for a number of automated deduction systems, including Automath, Nuprl, HOL and Coq [32, 19, 39, 8], among others. In addition to formalizing mathematics, these systems are widely used for the analysis and verification of computer programs. To do this, one must draw a connection between the programming language used and the language of type theory; however, these connections have typically been informal translations, diminishing the significance of the formal verification results.

Formal connections have been drawn in the work of Reynolds [86] and Harper and Mitchell [45], each of whom sought to use type-theoretic analysis to explain an entire programming language. Reynolds gave a type-theoretic interpretation of Idealized Algol, and Harper and Mitchell did the same for a simplified fragment of Standard ML. Recently, Harper and Stone [48] have given such an interpretation of full Standard ML (Revised) [72]. However, in each of these cases, the type theories used were not sufficiently rich to form a foundation for mathematical reasoning; for example, they were unable to express equality or induction principles. On the other hand, Kreitz [59] gave an embedding of a fragment of Objective CAML [64] into the foundational type theory of Nuprl. However, this fragment omitted some important constructs, such as recursion and modules.

The difficulty has been that the same features of foundational type theories that make them so expressive also highly constrain their semantics, thereby restricting the constructs that may be introduced into them. For example, as I will discuss below, the existence of induction principles precludes the typing of *fix* that is typical in programming languages. In this chapter I show how to give a semantics to practical programming languages in foundational type theory. In particular, I give an embedding of λ^K into the Nuprl type theory. This embedding is simple and syntax-directed, which has been vital for its use in practical reasoning.

The applications of type-theoretic semantics are not limited to formal reasoning about programs. Using such a semantics it can be considerably easier to prove desirable properties about a programming language, such as type preservation, than with other means. We will see two such examples in Section 3.4. The usefulness of such semantics is also not limited to one particular programming language at a time. If two languages are given type-theoretic semantics, then one may use type theory to show relationships between the two, and when the semantics are simple, those relationships need be no more complicated than the inherent differences between the two. This is particularly useful in the area of type-directed compilation [93, 75, 61, 43, 77]. The process of type-directed compilation consists of (in part) translations between various typed intermediate languages. Embedding each into a common foundational type theory provides an ideal framework for showing the invariance of program meaning throughout the compilation process.

This semantics is also useful even if one ultimately desires a semantics in some framework other than type theory. Martin-Löf type theory is closely tied to a structured operational semantics and has denotational models in many frameworks including partial equivalence relations [5, 40], set theory [54] and domain theory [87, 81, 80]. Thus, foundational type theory may be used as a "semantic intermediate language."

3.1 A Computational Denotational Semantics

My main motivation for a type-theoretic semantics has been to draw formal connections between programming languages and type theory, thereby making type theory a powerful tool for reasoning about languages and programs without sacrificing any formality. However, a type-theoretic semantics is also valuable in its own right as mathematical model of a programming language.

Most programming language semantics are either operational or denotational. A typical operational semantics is specified by giving an evaluation relation on program terms (or an evaluation relation on some abstract machine along with a translation into that machine). Operational semantics have the advantage that they draw direct connections to computation, and explaining how programs compute is one of the prime functions of a semantics. However, operational semantics are typically rather brittle; a slight addition or change to an operational semantics often requires reworking all proofs of properties of that semantics.

In contrast, a denotational semantics specifies, for every program term, a mathematical object that the program term denotes. Typically a term's denotational is determined by composing in some way the denotations of its subterms. The compositionality of denotational semantics usually makes them more robust to change than a typical operational semantics. Furthermore, the equational theory of a denotational semantics is easier to work with since it derives directly from the mathematical objects, without need for an intermediating evaluation relation, and without needing to consider any surrounding context. However, the connection to computation in a typical operational semantics (although present) is much more remote than with an operational semantics.

A denotational semantics in type theory provides the advantages of both a denotational semantics. The type-theoretic semantics I present *is* denotational, and accrues all the attendant advantages of a denotational semantics, but type theory is in essence a programming language itself (with its own operational semantics), so this semantics also draws a strong connection to computation.

3.2 The Language of Type Theory

I begin with an informal overview of the programming features of the Nuprl type theory. It is primarily those programming features that I will use in the embedding. The logic of types is obtained through the propositions-as-types isomorphism [51], but this will not be critical to our purposes in this chapter. I present and discuss the type theory in detail in Chapter 4.

As base types, the theory contains integers (denoted by $\mathbb{Z}$), booleans[1] (denoted by $\mathbb{B}$), strings (denoted by *Atom*), the empty type *Void*, the trivial type *Unit*, and the type *Top* (which

[1]Booleans are actually defined in terms of the disjoint union type.

contains every well-formed term, and in which all well-formed terms are equal). Complex types are built from the base types using various type constructors such as disjoint unions (denoted by $T_1 + T_2$), dependent products[2] (denoted by $\Sigma x{:}T_1.T_2$) and dependent function spaces (denoted by $\Pi x{:}T_1.T_2$). When x does not appear free in T_2, we write $T_1 \times T_2$ for $\Sigma x{:}T_1.T_2$ and $T_1 \rightarrow T_2$ for $\Pi x{:}T_1.T_2$.

This gives an account of most of the familiar programming constructs other than polymorphism. To handle polymorphism we want to have functions that can take types as arguments. These can be typed with the dependent types discussed above if one adds a type of all types. Unfortunately, a single type of all types is known to make the theory inconsistent [36, 27, 70, 52], so instead the type theory includes a predicative hierarchy of universes, $\mathbb{U}_1, \mathbb{U}_2, \mathbb{U}_3$, etc. The universe $\mathbb{U}_1$ contains all types built up from the base types only, and the universe $\mathbb{U}_{i+1}$ contains all types built up from the base types and the universes $\mathbb{U}_1, \ldots, \mathbb{U}_i$. In particular, no universe is a member of itself.

Unlike λ^K, which has distinct syntactic classes for kinds, type constructors and terms, Nuprl has only one syntactic class for all expressions. As a result, types are first class citizens and may be computed just as any other term. For example, the expression *if b then $\mathbb{Z}$ else Top* (where b is a boolean expression) is a valid type. Evaluation is call-by-name, but these constructions may also be used in a call-by-value type theory with little modification.

To state the soundness of the embedding of λ^K, we will require two assertions from the logic of types. These are equality, denoted by $t_1 = t_2\ in\ T$, which states that the terms t_1 and t_2 are equal as members of type T, and subtyping, denoted by $T_1 \preceq T_2$, which states that every member of type T_1 is in type T_2 (and that terms equal in T_1 are equal in T_2). A membership assertion, denoted by $t\ in\ T$, is defined as $t = t\ in\ T$. The basic judgement in Nuprl is $H \vdash_\nu P$, which states that in context H (which contains hypotheses and declarations of variables) the proposition P is true. Often the proposition P will be an assertion of equality or membership in a type.

The basic operators discussed above are summarized in Figure 3.1. Note that the lambda abstractions of Nuprl are untyped, unlike those of λ^K. In addition to the operators discussed here, the type theory contains some other less familiar type constructors: the partial type, set type and very dependent function type. In order to better motivate these type constructors, we defer discussion of them until their point of relevance. The dynamic semantics of all the type-theoretic operators is given in Figure 4.3.

3.2.1 Domain and Category Theory

Some of the mechanisms I use in the following section to give the type-theoretic semantics of λ^K will look similar to mechanisms used to give programming language semantics in domain or category theory. This is not coincidence; the three theories are closely related and some of the mechanisms I use (such as the partial types of Section 3.3.2) are adapted from the folklore of domain theory.

However, domain theory and category theory are not interchangeable with type theory for the purposes in this dissertation. Type theory provides the highest degree of structure of the three theories, domain theory hides some structure in the interest of greater abstraction and generality, and category theory provides the least structure and the most abstraction and generality. Thus, one can easily construct a domain or a category based on the Nuprl type

[2] These are sometimes referred to in the literature as dependent sums, but I prefer the terminology to suggest the connection to the non-dependent type $T_1 \times T_2$.

	Type Formation	Introduction	Elimination
universe i	$\mathbb{U}_i$ (for $i \geq 1$)	type formation operators	
disjoint union	$T_1 + T_2$	$inj_1(e)$ $inj_2(e)$	$case(e, x_1.e_1, x_2.e_2)$
function space	$\Pi x{:}T_1.T_2$	$\lambda x.e$	$e_1 e_2$
product space	$\Sigma x{:}T_1.T_2$	$\langle e_1, e_2 \rangle$	$\pi_1(e)$ $\pi_2(e)$
integers	$\mathbb{Z}$	$\ldots, -1, 0, 1, 2, \ldots$	assorted operations
booleans	$\mathbb{B}$	$true, false$	if-then-else
atoms	$Atom$	string literals	equality test ($=_A$)
void	$Void$		
unit	$Unit$	$\star$	
top	Top		

Figure 3.1: Type Theory Syntax

theory, but one cannot easily work backwards (although such models of type theory do exist [87, 81, 80]).

The structure of the Nuprl type theory stems from the fact that Nuprl provides direct access to the primitives of computation and the types of Nuprl speak directly of the computational behavior of terms. This structure is essential for achieving my goal of establishing a direct computational significance for this semantics (recall Section 3.1). This structure also allows type theory to address directly issues that pose significant challenges for domain theory (and that are not directly meaningful in category theory). For example, in type theory we may easily include or exclude divergent terms from types, allowing us to easily distinguish between partial or total functions, and, more importantly, allowing us to use induction properties that are invalid if one includes divergent terms.

3.3 A Type-Theoretic Semantics

I present the embedding of λ^K into type theory in three parts. In the first part I begin by giving embeddings for most of the basic type and term operators. These embeddings are uniformly straightforward. Second, I examine what happens when the embedding is expanded to include *fix*. There we will find it necessary to modify some of the original embeddings of the basic operators. In the third part I complete the semantics by giving embeddings for the kind-level constructs of λ^K. The complete embedding is summarized in Figures 3.3 through 3.6.

The embedding itself could be formulated in type theory, leaving to metatheory only the trivial task of encoding the abstract syntax of the programming language. Were this done, the theorems of Section 3.4 could be proven within the framework of type theory. For simplicity, however, I will state the embedding and theorems in metatheory.

3.3.1 Basic Embedding

The embedding is defined as a syntax-directed mapping (denoted by $[\![\cdot]\!]$) of λ^K expressions to terms of type theory. Recall that in Nuprl all expressions are terms; in particular, types are terms and may be computed just as any other term. Many λ^K expressions are translated directly into type theory:

$$\begin{array}{rcl}
[\![x]\!] & \stackrel{\text{def}}{=} & x \\
[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
[\![\lambda x{:}c.e]\!] & \stackrel{\text{def}}{=} & \lambda x.[\![e]\!] \\
[\![e_1 e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!][\![e_2]\!] \\
[\![c_1 \rightarrow c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \rightarrow [\![c_2]\!]
\end{array}$$

Nothing happens here except that the types are stripped out of lambda abstractions to match the syntax of Nuprl. Functions at the type constructor level are equally easy to embed, but I defer discussion of them until Section 3.3.3.

Since the type theory does not distinguish between functions taking term arguments and functions taking type arguments, polymorphic functions may be embedded just as easily, although a dependent type is required to express the dependency of c on α in the polymorphic type $\forall\alpha{:}\kappa.c$:

$$\begin{array}{rcl}
[\![\Lambda\alpha{:}\kappa.e]\!] & \stackrel{\text{def}}{=} & \lambda\alpha.[\![e]\!] \\
[\![e[c]]\!] & \stackrel{\text{def}}{=} & [\![e]\!][\![c]\!] \\
[\![\forall\alpha{:}\kappa.c]\!] & \stackrel{\text{def}}{=} & \Pi\alpha{:}[\![\kappa]\!].[\![c]\!]
\end{array}$$

Just as the type was stripped out of the lambda abstraction above, the kind is stripped out of the polymorphic abstraction. The translation of the polymorphic function type above makes use of the embedding of kinds, but, except for the elementary kind *Type*, I defer discussion of the embedding of kinds until Section 3.3.3. The kind Type_i, which contains level-i types, is embedded as the universe containing level-i types:

$$[\![\mathit{Type}_i]\!] \stackrel{\text{def}}{=} \mathbb{U}_i$$

Records and Disjoint Unions A bit more delicate than the above, but still fairly simple, is the embedding of records. Field labels are taken to be members of type *Atom*, and then records are viewed as functions that map field labels to the contents of the corresponding fields. For example, the record $\{\mathtt{x} = 1, \mathtt{f} = \lambda x{:}\mathit{int}.x\}$, which has type $\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \rightarrow \mathit{int}\}$, is embedded as

$$\lambda a.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ 1\ \mathit{else}\ \mathit{if}\ a =_A \mathtt{f}\ \mathit{then}\ \lambda x.x\ \mathit{else}\ \star$$

where $a =_A a'$ is the equality test on atoms, which returns a boolean when a and a' are atoms.

Since the type of this function's result depends upon its argument, this function must be typed using a dependent type:

$$\Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ \mathbb{Z}\ \mathit{else}\ \mathit{if}\ a =_A \mathtt{f}\ \mathit{then}\ \mathbb{Z} \rightarrow \mathbb{Z}\ \mathit{else}\ \mathit{Top}$$

In general, records and record types are embedded as follows:

$$
\begin{array}{rcl}
[\![\{\ell_1 = e_1, \ldots, \ell_n = e_n\}]\!] & \stackrel{\text{def}}{=} & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![e_1]\!] \\
& & \quad \vdots \\
& & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![e_n]\!] \\
& & \quad \mathit{else}\ \star \\
[\![\pi_\ell(e)]\!] & \stackrel{\text{def}}{=} & [\![e]\!]\ \ell \\
[\![\{\ell_1 : c_1, \ldots, \ell_n : c_n\}]\!] & \stackrel{\text{def}}{=} & \Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
& & \qquad \mathit{else\ Top}
\end{array}
$$

Note that this embedding validates the desired subtyping relationship on records. Since $\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \to \mathit{int}\} \preceq \{\mathtt{x} : \mathit{int}\}$, we would like the embedding to respect the subtyping relationship: $[\![\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \to \mathit{int}\}]\!] \preceq [\![\{\mathtt{x} : \mathit{int}\}]\!]$. Fortunately this is the case, since every type is a subtype of *Top*, and in particular the part of the type relating to the omitted field, $\mathit{if}\ a = \mathtt{f}\ \mathit{then}\ \mathbb{Z} \to \mathbb{Z}\ \mathit{else\ Top}$, is a subtype of *Top*. The use of a type *Top* to catch extra labels is essential for subtyping to work properly makes for a particularly elegant embedding of records, but it is not essential. In the absence of *Top* one could produce a slightly less elegant embedding by restricting the domain to exclude undesired labels using a set type (Section 3.3.3).

Disjoint unions are handled in a similar manner. The injection term $\mathit{inj}_{\mathtt{x}}(1)$ is embedded as the pair $\langle \mathtt{x}, 1\rangle$ of its label and its argument. The types of this term include the sum type $\langle \mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \to \mathit{int}\rangle$, which is embedded using a dependent type as:

$$\Sigma a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ \mathbb{Z}\ \mathit{else\ if}\ a =_A \mathtt{f}\ \mathit{then}\ \mathbb{Z} \to \mathbb{Z}\ \mathit{else\ Void}$$

In general, disjoint unions are embedded as follows:

$$
\begin{array}{rcl}
[\![\langle \ell_1 : c_1, \ldots, \ell_n : c_n\rangle]\!] & \stackrel{\text{def}}{=} & \Sigma a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
& & \qquad \mathit{else\ Void} \\
[\![\mathit{inj}_\ell(e)]\!] & \stackrel{\text{def}}{=} & \langle \ell, e\rangle \\
[\![\,\mathit{case}(e, \ell_1 \triangleright x_1.e_1, \ldots, \ell_n \triangleright x_n.e_n)]\!] & \stackrel{\text{def}}{=} & \mathit{if}\ \pi_1(x) =_A \ell_1\ \mathit{then}\ [\![e_1]\!][\pi_2(x)/x_1] \\
& & \vdots \\
& & \mathit{else\ if}\ \pi_1(x) =_A \ell_n\ \mathit{then}\ [\![e_n]\!][\pi_2(x)/x_n] \\
& & \mathit{else}\ \star
\end{array}
$$

Again, this relation validates the desired subtyping relationship.

3.3.2 Embedding Recursion

A usual approach to typing general recursive definitions of functions, and the one used in λ^K, is to add a *fix* construct with the typing rule:

$$\frac{H \vdash_\nu e\ \mathit{in}\ T \to T}{H \vdash_\nu \mathit{fix}(e)\ \mathit{in}\ T} \qquad \text{(wrong)}$$

In effect, this adds recursively defined (and possibly divergent) terms to existing types. Unfortunately, such a broad fixpoint rule makes Martin-Löf type theories inconsistent because of the presence of induction principles. An induction principle on a type specifies the membership of that type; for example, the standard induction principle on the natural numbers specifies that every natural number is either zero or some finite iteration of successor on zero. The ability to add divergent elements to a type would violate the specification implied by that type's induction rule.

One simple way to derive an inconsistency from the above typing rule uses the simplest induction principle, induction on the empty type *Void*. The induction principle for *Void* indirectly specifies that it has no members:

$$\frac{H \vdash_\nu e \ \mathit{in}\ \mathit{Void}}{H \vdash_\nu e \ \mathit{in}\ T}$$

However, it would be easy, using *fix*, to derive a member of *Void*: the identity function can be given type $\mathit{Void} \to \mathit{Void}$, so $\mathit{fix}(\lambda x.x)$ would have type *Void*. Invoking the induction principle, $\mathit{fix}(\lambda x.x)$ would be a member of every type and, by the propositions-as-types isomorphism, would be a proof of every proposition. It is also worth noting that this inconsistency does not stem from the fact that *Void* is an empty type; similar inconsistencies may be derived (with a bit more work) for almost every type, including function types (to which the *fix* rule of λ^K is restricted).

It is clear, then, that *fix* cannot be used to define new members of the basic types. How then can recursive functions be typed? The solution is to add a new type constructor for *partial types* [24, 25, 92]. For any type T, the partial type $\overline{T}$ is a supertype of T that contains all the elements of T and also all divergent terms. (A *total* type is one that contains only convergent terms.) The induction principles on $\overline{T}$ are different than those on T, so we can safely type *fix* with the rule:[3]

$$\frac{H \vdash_\nu e \ \mathit{in}\ \overline{T} \to \overline{T} \quad H \vdash_\nu T \ \text{admiss}}{H \vdash_\nu \mathit{fix}(e) \ \mathit{in}\ \overline{T}}$$

We use partial types to interpret the possibly non-terminating computations of λ^K. When (in λ^K) a term e has type τ, the embedded term $[\![e]\!]$ will have type $\overline{[\![\tau]\!]}$. Moreover, if e is valuable, then $[\![e]\!]$ can still be given the stronger type $[\![\tau]\!]$. Before we can embed *fix* we must re-examine the embedding of function types. In Nuprl, partial functions are viewed as functions with partial result types:[4]

$$\begin{aligned}
[\![c_1 \to c_2]\!] &\stackrel{\text{def}}{=} [\![c_1]\!] \to \overline{[\![c_2]\!]} \\
[\![c_1 \Rightarrow c_2]\!] &\stackrel{\text{def}}{=} [\![c_1]\!] \to [\![c_2]\!] \\
[\![\forall\alpha{:}\kappa.c]\!] &\stackrel{\text{def}}{=} \Pi\alpha{:}[\![\kappa]\!].[\![c]\!]
\end{aligned}$$

Note that, as desired, $[\![\tau_1 \Rightarrow \tau_2]\!] \preceq [\![\tau_1 \to \tau_2]\!]$, since $[\![\tau_2]\!] \preceq \overline{[\![\tau_2]\!]}$. If partial polymorphic functions were included in λ^K, they would be embedded as $\Pi\alpha{:}[\![\kappa]\!].\overline{[\![c]\!]}$.

Now suppose we wish to *fix* the function f which (in λ^K) has type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$, and suppose, for simplicity only, that f is valuable. Then $[\![f]\!]$ has type $([\![\tau_1]\!] \to \overline{[\![\tau_2]\!]}) \to \overline{[\![\tau_1]\!] \to \overline{[\![\tau_2]\!]}}$.

[3]The second subgoal, that the type T be *admissible,* is a technical condition related to the notion of admissibility in LCF [38]. All the types used in the embedding are admissible, so I ignore the admissibility condition in the discussion of this chapter. Admissibility is discussed further in Chapter 4 and is examined in detail in Chapter 5.

[4]This terminology can be somewhat confusing. A total type is one that contains only convergent expressions. The partial *function* type $T_1 \to \overline{T}_2$ contains functions that *return* possibly divergent elements, but those functions themselves converge, so a partial function type is a total type.

This type does not quite fit the *fix* typing rule, which requires the domain type to be partial, so we must coerce $\llbracket f \rrbracket$ to a fixable type. We do this by eta-expanding $\llbracket f \rrbracket$ to gain access to its argument (call it g) and then eta-expanding that argument g:

$$\lambda g.\llbracket f \rrbracket(\lambda x.g\,x)\ \mathit{in}\ \overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}} \to \overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}}$$

Eta-expanding g ensures that it terminates (since $\lambda x.g\,x$ is a canonical form), changing its type from $\overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}}$ to $\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}$. The former type is required by the *fix* rule, but the latter type is expected by $\llbracket f \rrbracket$. Since the coerced $\llbracket f \rrbracket$ fits the *fix* typing rule, we get that $\mathit{fix}(\lambda g.\llbracket f \rrbracket(\lambda x.g\,x))$ has type $\overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}}$, as desired. Thus we may embed the *fix* construct as:

$$\llbracket \mathit{fix}_c(e) \rrbracket \ \stackrel{\text{def}}{=}\ \mathit{fix}(\lambda g.\llbracket e \rrbracket(\lambda x.g\,x))$$

Strictness In λ^K, a function may be applied to a possibly divergent argument, but in my semantics functions expect their arguments to be convergent. Therefore we must change the embedding of application to compute function arguments to canonical form before applying the function.[5] This is done using the sequencing construct $\mathit{let}\ x = e_1\ \mathit{in}\ e_2$ which evaluates e_1 to canonical form e_1' and then reduces to $e_2[e_1'/x]$. The sequence term diverges if e_1 or e_2 does and allows x be given a total type:

$$\frac{H \vdash_\nu e_1\ \mathit{in}\ \overline{T}_2 \quad H; x{:}T_2 \vdash_\nu e_2\ \mathit{in}\ \overline{T_1}}{H \vdash_\nu \mathit{let}\ x = e_1\ \mathit{in}\ e_2\ \mathit{in}\ \overline{T_1}}$$

Application is then embedded in the expected way:

$$\llbracket e_1 e_2 \rrbracket \ \stackrel{\text{def}}{=}\ \mathit{let}\ x = \llbracket e_2 \rrbracket\ \mathit{in}\ \llbracket e_1 \rrbracket\,x$$

A final issue arises in regard to records and disjoint unions. In the embedding of Section 3.3.1, the record $\{\ell = e\}$ would terminate even if e diverges. This would be unusual in a call-by-value programming language, so we need to ensure that each member of a record is evaluated:

$$\begin{array}{rcl} \llbracket \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \rrbracket & \stackrel{\text{def}}{=} & \mathit{let}\ x_1 = \llbracket e_1 \rrbracket\ \mathit{in} \\ & & \vdots \\ & & \mathit{let}\ x_n = \llbracket e_n \rrbracket\ \mathit{in} \\ & & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\ & & \quad\ \vdots \\ & & \quad\ \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\ & & \quad\ \mathit{else}\ \star \end{array}$$

A similar change must be also be made for disjoint unions:

$$\llbracket \mathit{inj}_\ell(e) \rrbracket \ \stackrel{\text{def}}{=}\ \mathit{let}\ x = \llbracket e \rrbracket\ \mathit{in}\ \langle \ell, x \rangle$$

[5]Polymorphic functions are unaffected because all type expressions converge (Corollary 3.4).

3.3.3 Embedding Kinds

The kind structure of λ^K contains three first-order kind constructors. We have already seen the embedding of the kind *Type*; remaining are the power and singleton kinds. Each of these kinds represents a collection of types, so each will be embedded as something similar to a universe, but unlike the kind $Type_i$, which includes all types of the indicated level, the power and singleton kinds wish to exclude certain undesirable types. The power kind $\mathcal{P}_i(\tau)$ contains only subtypes of τ and the singleton kind $\mathcal{S}_i(\tau)$ contains only types that are equal to τ; other types must be left out.

The mechanism for achieving this exclusion is the *set type* [21]. If S is a type and $P[\cdot]$ is a predicate over S, then the set type $\{z : S \mid P[z]\}$ contains all elements z of S such that $P[z]$ is true. With this type, we can embed the power and singleton kinds as:[6]

$$\begin{array}{rcl}
[\![\mathcal{P}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T \preceq [\![c]\!] \wedge [\![c]\!] \ \mathit{in}\ \mathbb{U}_i\} \\
[\![\mathcal{S}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T = [\![c]\!] \ \mathit{in}\ \mathbb{U}_i\}
\end{array}$$

Among the higher-order type constructors, functions at the type constructor level and their kinds are handled just as at the term level, except that function kinds are permitted to have dependencies but need not deal with partiality or strictness:

$$\begin{array}{rcl}
[\![\lambda\alpha{:}\kappa.c]\!] & \stackrel{\text{def}}{=} & \lambda\alpha.[\![c]\!] \\
[\![c_1[c_2]]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!][\![c_2]\!] \\
[\![\Pi\alpha{:}\kappa_1.\kappa_2]\!] & \stackrel{\text{def}}{=} & \Pi\alpha{:}[\![\kappa_1]\!].[\![\kappa_2]\!]
\end{array}$$

Dependent Record Kinds For records at the type constructor level, the embedding of the records themselves is analogous to those at the term level (except that there is no issue of strictness):

$$\begin{array}{rcl}
[\![\{\ell_1 = c_1, \ldots, \ell_n = c_n\}]\!] & \stackrel{\text{def}}{=} & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
 & & \qquad \vdots \\
 & & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
 & & \qquad \mathit{else}\ \star \\
[\![\pi_\ell(c)]\!] & \stackrel{\text{def}}{=} & [\![c]\!]\,\ell
\end{array}$$

However, the embedding of this expression's kind is more complicated. This is because of the need to express dependencies among the fields of the dependent record kind. Recall that the embedding of a non-dependent record type already required a dependent type; to embed a dependent record type will require expressing even more dependency. Consider the dependent record kind $\{\ell \triangleright \alpha : Type_1, \ell' \triangleright \alpha' : \mathcal{P}_1(\alpha)\}$. We might naively attempt to encode this like the non-dependent record type as

$$\Pi a{:}Atom.\ \mathit{if}\ a =_A \ell\ \mathit{then}\ \mathbb{U}_1\ \mathit{else\ if}\ a =_A \ell'\ \mathit{then}\ \{T : \mathbb{U}_1 \mid T \preceq \alpha \wedge \alpha\ \mathit{in}\ \mathbb{U}_1\}\ \mathit{else}\ Top \quad \text{(wrong)}$$

but this encoding is not correct; the variable α is now unbound. We want α to refer to the contents of field ℓ. In the encoding, this means we want α to refer to the value returned by the function when applied to label ℓ. So we want a type of functions whose return type can depend not only upon their arguments but upon their own return values!

[6]The second clause in the embedding of the power kind ($[\![c]\!]\ \mathit{in}\ \mathbb{U}_i$) is used for technical reasons that require that well-formedness of $\mathcal{P}_i(\tau)$ imply that $\tau : Type_i$.

The type I will use for this embedding is Hickey's *very dependent function type* [49]. This type is a generalization of the dependent function type (itself a generalization of the ordinary function type) and like it, the very dependent function type's members are just lambda abstractions. The difference is in the specification of a function's return type. The type is denoted by $\{f \mid x{:}T_1 \to T_2\}$ where f and x are binding occurrences that may appear free in T_2 (but not in T_1).

As with the dependent function type, x stands for the function's argument, but the additional variable f refers to the function itself. A function g belongs to the type $\{f \mid x{:}T_1 \to T_2\}$ if g takes an argument from T_1 (call it t) and returns a member of $T_2[t, g/x, f]$.[7]

For example, the kind $\{\ell \triangleright \alpha : \mathit{Type}_1, \ell' \triangleright \alpha' : \mathcal{P}_1(\alpha)\}$ discussed above is encoded as a very dependent function type as:

$$\{f \mid a{:}\mathit{Atom} \to \mathit{if}\ a =_A \ell\ \mathit{then}\ \mathbb{U}_1\ \mathit{else\ if}\ a =_A \ell'\ \mathit{then}\ \{T : \mathbb{U}_1 \mid T \preceq f\,\ell \wedge f\,\ell\ \mathit{in}\ \mathbb{U}_1\}\ \mathit{else}\ \mathit{Top}\}$$

To understand where this type constructor fits in with the more familiar type constructors, consider the "type triangle" shown in Figure 3.2. On the right are the non-dependent type constructors and in the middle are the dependent type constructors. Arrows are drawn from type constructors to weaker ones that may be implemented with them. Horizontal arrows indicate when a weaker constructor may be obtained by dropping a possible dependency from a stronger one; for example, the function type $T_1 \to T_2$ is a degenerate form of the dependent function type $\Pi x{:}T_1.T_2$ where the dependent variable x is not used in T_2. Diagonal arrows indicate when a weaker constructor may be implemented with a stronger one by performing case analysis on a boolean; for example, the disjoint union type $T_1 + T_2$ is equivalent to the type $\Sigma b{:}\mathbb{B}.\ \mathit{if}\ b\ \mathit{then}\ T_1\ \mathit{else}\ T_2$.[8]

If we ignore the very dependent function type, the type triangle illustrates how the basic type constructors may be implemented by the dependent function and dependent product types. The very dependent function type completes this picture: the dependent function is a degenerate form where the f dependency is not used, and the dependent product may be implemented by switching on a boolean. Thus, the very dependent function type is a single unified type constructor from which all the basic type constructors may be constructed.

In general, dependent record kinds are encoded using a very dependent function type as follows:

$$
\begin{aligned}
[\![\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}]\!] \ \stackrel{\text{def}}{=}\ \ & \{f \mid a{:}\mathit{Atom} \to \\
& \quad \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\kappa_1]\!] \\
& \quad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\kappa_2]\!][f\,\ell_1/\alpha_1] \\
& \quad \vdots \\
& \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
& \qquad [\![\kappa_n]\!][f\,\ell_1 \cdots f\,\ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
& \quad \mathit{else}\ \mathit{Top}\}
\end{aligned}
$$

[7] To avoid the apparent circularity, in order for $\{f \mid x{:}T_1 \to T_2\}$ to be well-formed we require that T_2 may only use the result of f when applied to elements of T_1 that are less than x with regard to some well-founded order. This restriction will not be a problem for this embedding because the order in which field labels appear in a dependent record kind is a perfectly good well-founded order.

[8] By switching on a label, instead of a boolean, record types and tagged variant types could be implemented and placed along the diagonals as well.

$$
\begin{tikzcd}
\{f \mid x{:}T_1 \to T_2\} \arrow[r] \arrow[dr] & \Pi x{:}T_1.T_2 \arrow[r] \arrow[dr] & T_1 \to T_2 \\
& \Sigma x{:}T_1.T_2 \arrow[r] \arrow[dr] & T_1 \times T_2 \\
& & T_1 + T_2
\end{tikzcd}
$$

Figure 3.2: The Type Triangle

3.3.4 Embedding Modules

As discussed in Section 2.4.1, λ^K uses a phase-splitting interpretation of modules, where modules (including higher-order modules) are considered to consist of two components: a compile-time component and a run-time component. This is reflected in the type-theoretic semantics by an embedding that explicitly phase-splits modules. The technique used is derived from Harper, *et al.* [46].

The embeddings for modules and signatures are given by two syntax-directed mappings, $\llbracket \cdot \rrbracket_{\mathrm{c}}$ and $\llbracket \cdot \rrbracket_{\mathrm{r}}$, one for the compile-time component of the given expression and one for the run time component. Given these, the embedding of a module is a pair of the compile-time and run-time components:

$$\llbracket m \rrbracket \quad \stackrel{\mathrm{def}}{=} \quad \langle \llbracket m \rrbracket_{\mathrm{c}}, \llbracket m \rrbracket_{\mathrm{r}} \rangle$$

In signatures, the types of run-time fields may depend upon a compile-time member, as in the signature $\{\texttt{foo} \triangleright s_{\mathrm{foo}} : \langle \mathit{Type} \rangle, \texttt{bar} : \langle\!\langle \mathit{ext}(s_{\mathrm{foo}}) \rangle\!\rangle\}$ (corresponding to the KML module `sig tycon foo : type val bar : foo end`). Consequently, the embedding of a signature is the dependent product of the compile-time component and the run-time component. The run-time component's embedding is a function that takes as an argument the compile-time member on which it depends. In the embedding of the full signature, that function is applied to the variable standing for the compile-time member:

$$\llbracket \sigma \rrbracket \quad \stackrel{\mathrm{def}}{=} \quad \Sigma v{:}\llbracket \sigma \rrbracket_{\mathrm{c}}.\llbracket \sigma \rrbracket_{\mathrm{r}} v$$

The embeddings of basic modules and signatures are simple. The run-time component is trivial for $\langle \kappa \rangle$ signatures and $\langle c \rangle$ modules, and the compile-time component is trivial for $\langle\!\langle c \rangle\!\rangle$ signatures and $\langle\!\langle e \rangle\!\rangle$. Module variables s are split into separate variables s_{c} and s_{r} for each component.

$$
\begin{array}{rcll}
[\![\langle c \rangle]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![c]\!] & \text{(module compile-time component)} \\
[\![\langle c \rangle]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \star & \text{(module run-time component—trivial)} \\
[\![\langle \kappa \rangle]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![\kappa]\!] & \text{(signature compile-time component)} \\
[\![\langle \kappa \rangle]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \lambda v.\mathit{Top} & \text{(signature run-time component—trivial)} \\
[\![\langle\!\langle e \rangle\!\rangle]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \star & \text{(module compile-time component—trivial)} \\
[\![\langle\!\langle e \rangle\!\rangle]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & [\![e]\!] & \text{(module run-time component)} \\
[\![\langle\!\langle c \rangle\!\rangle]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \mathit{Top} & \text{(signature compile-time component—trivial)} \\
[\![\langle\!\langle c \rangle\!\rangle]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \lambda v.[\![c]\!] & \text{(signature run-time component)} \\
[\![s]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & s_{\mathrm{c}} & \text{(module compile-time component)} \\
[\![s]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & s_{\mathrm{r}} & \text{(module run-time component)}
\end{array}
$$

For each of the signatures above it is impossible for there to be any dependency of the run-time component on the compile-time component, since for $\langle \kappa \rangle$ there is no nontrivial run-time to depend on anything, and for $\langle\!\langle c \rangle\!\rangle$ there is no nontrivial compile-time for anything to depend on. As a result, the variable v is ignored in each case.

Functors For function modules and signatures, everything looks familiar in the compile-time component, where the run-time material is ignored:

$$
\begin{array}{rcl}
[\![\lambda s{:}\sigma.m]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \lambda s_{\mathrm{c}}.[\![m]\!]_{\mathrm{c}} \\
[\![m_1 m_2]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![m_1]\!]_{\mathrm{c}}[\![m_2]\!]_{\mathrm{c}} \\
[\![\Pi s{:}\sigma_1.\sigma_2]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \Pi s_{\mathrm{c}}{:}[\![\sigma_1]\!]_{\mathrm{c}}.[\![\sigma_2]\!]_{\mathrm{c}}
\end{array}
$$

However, the run-time component may not similarly ignore the compile-time component, because of possible dependencies. Therefore, the run-time component abstracts over both the compile-time and run-time components of its argument:

$$
\begin{array}{rcl}
[\![\lambda s{:}\sigma.m]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \lambda s_{\mathrm{c}}.\,\lambda s_{\mathrm{r}}.\,[\![m]\!]_{\mathrm{r}} \\
[\![m_1 m_2]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \mathit{let}\ x = [\![m_2]\!]_{\mathrm{r}}\ \mathit{in}\ [\![m_1]\!]_{\mathrm{r}}[\![m_2]\!]_{\mathrm{c}}x
\end{array}
$$

The signature's run-time component, then, takes an argument v representing the compile-time component and returns a curried function type. The result type is the run-time component of the result signature and that depends on the compile-time component. Fortunately, the result's compile-time component is available (as $v\,s_{\mathrm{c}}$), since v maps the compile-time component of the argument to the compile-time component of the result.

$$
[\![\Pi s{:}\sigma_1.\sigma_2]\!]_{\mathrm{r}} \stackrel{\mathrm{def}}{=} \lambda v.\,\Pi s_{\mathrm{c}}{:}[\![\sigma_1]\!]_{\mathrm{c}}.\,[\![\sigma_1]\!]_{\mathrm{r}} s_{\mathrm{c}} \rightarrow [\![\sigma_2]\!]_{\mathrm{r}}(v\,s_{\mathrm{c}})
$$

Structures For dependent record modules and signatures, the compile-time component looks like dependent record kinds and the type constructor records that belong to them:

$$
\begin{array}{rcl}
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \lambda a.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![m_1]\!]_{\mathrm{c}} \\
& & \quad \vdots \\
& & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![m_n]\!]_{\mathrm{c}} \\
& & \quad \mathit{else}\ \star \\
[\![\pi_\ell(m)]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{c}}\ell \\
[\![\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \{f \mid a{:}\mathit{Atom} \rightarrow \\
& & \qquad \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\sigma_1]\!]_{\mathrm{c}} \\
& & \qquad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{c}}[f\,\ell_1/s_{1\mathrm{c}}] \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
& & \qquad\quad [\![\sigma_n]\!]_{\mathrm{c}}[f\,\ell_1 \cdots f\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{(n-1)\mathrm{c}}] \\
& & \qquad \mathit{else}\ \mathit{Top}\}
\end{array}
$$

The run-time component of modules looks much like the embedding of record terms (as with those, a series of opening lets is necessary to ensure strictness):

$$
\begin{array}{rcl}
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \mathit{let}\ x_1 = [\![m_1]\!]_{\mathrm{r}}\ \mathit{in} \\
& & \vdots \\
& & \mathit{let}\ x_n = [\![m_n]\!]_{\mathrm{r}}\ \mathit{in} \\
& & \lambda a.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
& & \quad \vdots \\
& & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
& & \quad \mathit{else}\ \star \\
& & (\text{where } x_i \text{ does not appear free in } m_i) \\
[\![\pi_\ell(m)]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{r}}\ell
\end{array}
$$

The run-time component of signatures is also familiar, except that it must deal with dependencies on the compile-time component. Again, the run-time component takes an argument v representing the compile-time component. The run-time component $[\![\sigma_i]\!]_{\mathrm{r}}$ of each field is applied to the compile-time component of that field, which is $v\,\ell_i$. Also, each field may have dependencies on the compile-time components of *earlier* fields. These dependencies will have been expressed by free occurrences of the variables s_{ic}, into which we substitute the corresponding compile-time components $v\,\ell_i$. It is worthwhile to note that these substitutions for s_{ic} are the only places where dependencies on the argument v are introduced.

$$
\begin{array}{l}
[\![\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}]\!]_{\mathrm{r}} \\
\quad \stackrel{\mathrm{def}}{=} \quad \lambda v.\, \Pi a{:}\mathit{Atom}.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\sigma_1]\!]_{\mathrm{r}}(v\,\ell_1) \\
\qquad\qquad\qquad\qquad\quad \mathit{if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{r}}(v\,\ell_2)[v\,\ell_1/s_{1\mathrm{c}}] \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
\qquad\qquad\qquad\qquad\qquad [\![\sigma_n]\!]_{\mathrm{r}}(v\,\ell_n)[v\,\ell_1 \cdots v\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{(n-1)\mathrm{c}}] \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \mathit{Top}
\end{array}
$$

$$
\begin{array}{rcl}
\llbracket \mathit{Type}_i \rrbracket & \stackrel{\mathrm{def}}{=} & \{T : \mathbb{U}_i \mid T \;\text{ admiss} \wedge T \;\text{ total}\} \\
\llbracket \Pi\alpha{:}\kappa_1.\kappa_2 \rrbracket & \stackrel{\mathrm{def}}{=} & \Pi\alpha{:}\llbracket \kappa_1 \rrbracket.\llbracket \kappa_2 \rrbracket \\
\llbracket \{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\} \rrbracket & \stackrel{\mathrm{def}}{=} & \{f \mid a{:}\mathit{Atom} \to \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket \kappa_1 \rrbracket \\
& & \qquad\qquad\qquad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ \llbracket \kappa_2 \rrbracket[f\ \ell_1/\alpha_1] \\
& & \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
& & \qquad\qquad\qquad\quad \llbracket \kappa_n \rrbracket[f\ \ell_1 \cdots f\ \ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
& & \qquad\qquad\qquad \mathit{else}\ \mathit{Top}\} \\
& & \text{(where } f, a \text{ do not appear free in } \kappa_i\text{)} \\
\llbracket \mathcal{P}_i(c) \rrbracket & \stackrel{\mathrm{def}}{=} & \{T : \mathbb{U}_i \mid T \preceq \llbracket c \rrbracket \wedge \llbracket c \rrbracket\ \mathit{in}\ \mathbb{U}_i \wedge T \;\text{ admiss}\} \\
& & \text{(where } T \text{ does not appear free in } c\text{)} \\
\llbracket \mathcal{S}_i(c) \rrbracket & \stackrel{\mathrm{def}}{=} & \{T : \mathbb{U}_i \mid T = \llbracket c \rrbracket\ \mathit{in}\ \mathbb{U}_i\} \\
& & \text{(where } T \text{ does not appear free in } c\text{)} \\
& & \\
\llbracket \alpha \rrbracket & \stackrel{\mathrm{def}}{=} & \alpha \\
\llbracket \lambda\alpha{:}\kappa.c \rrbracket & \stackrel{\mathrm{def}}{=} & \lambda\alpha.\llbracket c \rrbracket \\
\llbracket c_1[c_2] \rrbracket & \stackrel{\mathrm{def}}{=} & \llbracket c_1 \rrbracket \llbracket c_2 \rrbracket \\
\llbracket \{\ell_1 = c_1, \ldots, \ell_n = c_n\} \rrbracket & \stackrel{\mathrm{def}}{=} & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket c_1 \rrbracket \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ \llbracket c_n \rrbracket \\
& & \qquad \mathit{else}\ \star \\
& & \text{(where } a \text{ does not appear free in } c_i\text{)} \\
\llbracket \pi_\ell(c) \rrbracket & \stackrel{\mathrm{def}}{=} & \llbracket c \rrbracket\ \ell \\
\llbracket c_1 \to c_2 \rrbracket & \stackrel{\mathrm{def}}{=} & \llbracket c_1 \rrbracket \to \overline{\llbracket c_2 \rrbracket} \\
\llbracket c_1 \Rightarrow c_2 \rrbracket & \stackrel{\mathrm{def}}{=} & \llbracket c_1 \rrbracket \to \llbracket c_2 \rrbracket \\
\llbracket \forall\alpha{:}\kappa.c \rrbracket & \stackrel{\mathrm{def}}{=} & \Pi\alpha{:}\llbracket \kappa \rrbracket.\llbracket c \rrbracket \\
\llbracket \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \rrbracket & \stackrel{\mathrm{def}}{=} & \Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket c_1 \rrbracket \\
& & \qquad\qquad \vdots \\
& & \qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ \llbracket c_n \rrbracket \\
& & \qquad\qquad\quad \mathit{else}\ \mathit{Top} \\
& & \text{(where } a \text{ does not appear free in } c_i\text{)} \\
\llbracket \langle \ell_1 : c_1, \ldots, \ell_n : c_n \rangle \rrbracket & \stackrel{\mathrm{def}}{=} & \Sigma a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket c_1 \rrbracket \\
& & \qquad\qquad \vdots \\
& & \qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ \llbracket c_n \rrbracket \\
& & \qquad\qquad\quad \mathit{else}\ \mathit{Void} \\
& & \text{(where } a \text{ does not appear free in } c_i\text{)} \\
\llbracket\ \mathit{ext}(m) \rrbracket & \stackrel{\mathrm{def}}{=} & \llbracket m \rrbracket_{\mathrm{c}}
\end{array}
$$

Figure 3.3: Embedding Kinds and Types

$$
\begin{array}{rcl}
[\![x]\!] & \stackrel{\text{def}}{=} & x \\
[\![\lambda x{:}c.e]\!] & \stackrel{\text{def}}{=} & \lambda x.[\![e]\!] \\
[\![e_1 e_2]\!] & \stackrel{\text{def}}{=} & \mathit{let}\ x = [\![e_2]\!]\ \mathit{in}\ [\![e_1]\!]\, x \\
 & & \text{(where } x \text{ does not appear free in } e_1\text{)} \\
[\![\Lambda\alpha{:}\kappa.e]\!] & \stackrel{\text{def}}{=} & \lambda\alpha.[\![e]\!] \\
[\![e[c]]\!] & \stackrel{\text{def}}{=} & [\![e]\!][\![c]\!] \\
[\![\{\ell_1 = e_1, \ldots, \ell_n = e_n\}]\!] & \stackrel{\text{def}}{=} & \mathit{let}\ x_1 = [\![e_1]\!]\ \mathit{in} \\
 & & \vdots \\
 & & \mathit{let}\ x_n = [\![e_n]\!]\ \mathit{in} \\
 & & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
 & & \quad\vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
 & & \quad \mathit{else}\ \star \\
 & & \text{(where } x_i \text{ does not appear free in } e_j\text{)} \\
[\![\pi_\ell(e)]\!] & \stackrel{\text{def}}{=} & [\![e]\!]\, \ell \\
[\![\mathit{inj}_\ell(e)]\!] & \stackrel{\text{def}}{=} & \mathit{let}\ x = [\![e]\!]\ \mathit{in}\ \langle \ell, x\rangle \\
[\![\mathit{case}(e, \ell_1 \triangleright x_1.e_1, \ldots, \ell_n \triangleright x_n.e_n)]\!] & \stackrel{\text{def}}{=} & \mathit{let}\ x = [\![e]\!]\ \mathit{in} \\
 & & \mathit{if}\ \pi_1(x) =_A \ell_1\ \mathit{then}\ e_1[\pi_2(x)/x_1] \\
 & & \vdots \\
 & & \mathit{else\ if}\ \pi_1(x) =_A \ell_n\ \mathit{then}\ e_n[\pi_2(x)/x_n] \\
 & & \mathit{else}\ \star \\
 & & \text{(where } x \text{ does not appear free in } e_i\text{)} \\
[\![\mathit{fix}_c(e)]\!] & \stackrel{\text{def}}{=} & \mathit{fix}(\lambda g.[\![e]\!](\lambda x.g\, x)) \\
 & & \text{(where } g \text{ does not appear free in } e\text{)} \\
[\![\mathit{ext}(m)]\!] & \stackrel{\text{def}}{=} & [\![m]\!]_{\mathrm{r}}
\end{array}
$$

Figure 3.4: Embedding Terms

3.4 Properties of the Embedding

I conclude my presentation of the type-theoretic semantics of λ^K by examining some of the important properties of the semantics. We want the embedding to validate the intuitive meaning of the judgements of λ^K's static semantics. If κ_1 is a subkind of κ_2 then we want the embedded kind $[\![\kappa_1]\!]$ to be a subtype of $[\![\kappa_2]\!]$; if c_1 and c_2 are equal in kind κ, we want the embedded constructors $[\![c_1]\!]$ and $[\![c_2]\!]$ to be equal (in $[\![\kappa]\!]$); and if e has type τ we want $[\![e]\!]$ to have type $\overline{[\![\tau]\!]}$ (and $[\![\tau]\!]$ if e is valuable). Similar properties are desired for the module judgements. This is stated in Theorem 3.1:

Theorem 3.1 (Semantic Soundness) *For every λ^K context Γ, let $[\![\Gamma]\!]$ be defined as follows:*

$$
\begin{array}{rcl}
[\![\bullet]\!] & \stackrel{\text{def}}{=} & \epsilon \\
[\![\Gamma[\alpha : \kappa]]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!]; \alpha{:}[\![\kappa]\!] \\
[\![\Gamma[x : c]]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!]; x{:}[\![c]\!] \\
[\![\Gamma[s : \sigma]]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!]; s_{\mathrm{c}}{:}[\![\sigma]\!]_{\mathrm{c}}; s_{\mathrm{r}}{:}[\![\sigma]\!]_{\mathrm{r}} s_{\mathrm{c}}
\end{array}
$$

Then the following implications hold:

1. *If $\Gamma \vdash_K \kappa = \kappa'$ then $\mathit{level}(\kappa) = \mathit{level}(\kappa')$ and $[\![\Gamma]\!] \vdash_\nu [\![\kappa]\!] = [\![\kappa']\!]$ in $\mathbb{U}_{\mathit{level}(\kappa)+1}$*

$$
\begin{array}{rcl}
[\![\sigma]\!] & \stackrel{\text{def}}{=} & \Sigma v{:}[\![\sigma]\!]_{\mathrm{c}}.[\![\sigma]\!]_{\mathrm{r}} v \\
 & & \text{(where } v \text{ does not appear free in } \sigma) \\
[\![\langle\kappa\rangle]\!]_{\mathrm{c}} & \stackrel{\text{def}}{=} & [\![\kappa]\!] \\
[\![\langle\kappa\rangle]\!]_{\mathrm{r}} & \stackrel{\text{def}}{=} & \lambda v.\mathit{Top} \\
[\![\langle\!\langle c\rangle\!\rangle]\!]_{\mathrm{c}} & \stackrel{\text{def}}{=} & \mathit{Top} \\
[\![\langle\!\langle c\rangle\!\rangle]\!]_{\mathrm{r}} & \stackrel{\text{def}}{=} & \lambda v.[\![c]\!] \\
 & & \text{(where } v \text{ does not appear free in } c) \\
[\![\Pi s{:}\sigma_1.\sigma_2]\!]_{\mathrm{c}} & \stackrel{\text{def}}{=} & \Pi s_{\mathrm{c}}{:}[\![\sigma_1]\!]_{\mathrm{c}}.[\![\sigma_2]\!]_{\mathrm{c}} \\
[\![\Pi s{:}\sigma_1.\sigma_2]\!]_{\mathrm{r}} & \stackrel{\text{def}}{=} & \lambda v.\,\Pi s_{\mathrm{c}}{:}[\![\sigma_1]\!]_{\mathrm{c}}.\,[\![\sigma_1]\!]_{\mathrm{r}} s_{\mathrm{c}} \to [\![\sigma_2]\!]_{\mathrm{r}}(v\, s_{\mathrm{c}}) \\
 & & \text{(where } v, s \text{ do not appear free in } \sigma_i) \\
[\![\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}]\!]_{\mathrm{c}} & \stackrel{\text{def}}{=} & \{f \mid a{:}\mathit{Atom} \to \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\sigma_1]\!]_{\mathrm{c}} \\
 & & \qquad\qquad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{c}}[f\,\ell_1/s_{1\mathrm{c}}] \\
 & & \qquad\qquad \vdots \\
 & & \qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad\qquad\quad [\![\sigma_n]\!]_{\mathrm{c}}[f\,\ell_1 \cdots f\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{n-1\mathrm{c}}] \\
 & & \qquad\qquad \mathit{else}\ \mathit{Top}\} \\
 & & \text{(where } f, a \text{ do not appear free in } \sigma_i) \\
[\![\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}]\!]_{\mathrm{r}} & \stackrel{\text{def}}{=} & \lambda v.\,\Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\sigma_1]\!]_{\mathrm{r}}(v\,\ell_1) \\
 & & \qquad\qquad \mathit{if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{r}}(v\,\ell_2)[v\,\ell_1/s_{1\mathrm{c}}] \\
 & & \qquad\qquad \vdots \\
 & & \qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad\qquad\quad [\![\sigma_n]\!]_{\mathrm{r}}(v\,\ell_n) \\
 & & \qquad\qquad\qquad [v\,\ell_1 \cdots v\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{n-1\mathrm{c}}] \\
 & & \qquad\qquad \mathit{else}\ \mathit{Top}\} \\
 & & \text{(where } v, a \text{ do not appear free in } \sigma_i)
\end{array}
$$

Figure 3.5: Embedding Signatures

2. *If* $\Gamma \vdash_K \kappa \preceq \kappa'$ *then* $[\![\Gamma]\!] \vdash_\nu ([\![\kappa]\!]\ \mathit{in}\ \mathbb{U}_{level(\kappa)+1} \wedge [\![\kappa']\!]\ \mathit{in}\ \mathbb{U}_{level(\kappa')+1} \wedge [\![\kappa]\!] \preceq [\![\kappa']\!])$.
3. *If* $\Gamma \vdash_K c = c' : \kappa$ *then* $[\![\Gamma]\!] \vdash_\nu [\![c]\!] = [\![c']\!]\ \mathit{in}\ [\![\kappa]\!]$.
4. *If* $\Gamma \vdash_K c \preceq c'$ *then* $[\![\Gamma]\!] \vdash_\nu [\![c]\!] \preceq [\![c']\!]$.
5. *If* $\Gamma \vdash_K e : c$ *then* $[\![\Gamma]\!] \vdash_\nu [\![e]\!]\ \mathit{in}\ \overline{[\![c]\!]}$.
6. *If* $\Gamma \vdash_K e \downarrow c$ *then* $[\![\Gamma]\!] \vdash_\nu [\![e]\!]\ \mathit{in}\ [\![c]\!]$.
7. *If* $\Gamma \vdash_K \sigma \preceq \sigma'$ *then* $[\![\Gamma]\!] \vdash_\nu ([\![\sigma]\!]\ \mathit{in}\ \mathbb{U}_{level(\sigma)+1} \wedge [\![\sigma']\!]\ \mathit{in}\ \mathbb{U}_{level(\sigma')+1} \wedge [\![\sigma]\!] \preceq [\![\sigma']\!])$.
8. *If* $\Gamma \vdash_K m : \sigma$ *then* $[\![\Gamma]\!] \vdash_\nu ([\![m]\!]\ \mathit{in}\ \overline{[\![\sigma]\!]} \wedge [\![m]\!]_{\mathrm{c}}\ \mathit{in}\ [\![\sigma]\!]_{\mathrm{c}})$.
9. *If* $\Gamma \vdash_K m \downarrow \sigma$ *then* $[\![\Gamma]\!] \vdash_\nu [\![m]\!]\ \mathit{in}\ [\![\sigma]\!]$.

Proof

By induction on the derivations of the λ^K judgements.

We may observe two immediate consequences of the soundness theorem. One is the desirable property of type preservation: evaluation does not change the type of a program. Figure 4.3 gives a small-step evaluation relation for the Nuprl type theory (denoted by $t \mapsto t'$ when t evaluates in one step to t'). Type preservation of λ^K (Corollary 3.3) follows directly from soundness and type preservation of Nuprl (Proposition 3.2).

$$
\begin{array}{rcl}
[\![m]\!] & \stackrel{\mathrm{def}}{=} & \langle [\![m]\!]_{\mathrm{c}}, [\![m]\!]_{\mathrm{r}} \rangle \\
[\![s]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & s_{\mathrm{c}} \\
[\![s]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & s_{\mathrm{r}} \\
[\![\langle c \rangle]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![c]\!] \\
[\![\langle c \rangle]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \star \\
[\![\langle\!\langle e \rangle\!\rangle]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \star \\
[\![\langle\!\langle e \rangle\!\rangle]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & [\![e]\!] \\
[\![\lambda s{:}\sigma.m]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \lambda s_{\mathrm{c}}.[\![m]\!]_{\mathrm{c}} \\
[\![\lambda s{:}\sigma.m]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \lambda s_{\mathrm{c}}.\,\lambda s_{\mathrm{r}}.\,[\![m]\!]_{\mathrm{r}} \\
[\![m_1 m_2]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![m_1]\!]_{\mathrm{c}}[\![m_2]\!]_{\mathrm{c}} \\
[\![m_1 m_2]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \mathit{let}\ x = [\![m_2]\!]_{\mathrm{r}}\ \mathit{in}\ [\![m_1]\!]_{\mathrm{r}}[\![m_2]\!]_{\mathrm{c}}x \\
 & & \text{(where } x \text{ does not appear free in } m_1,\ m_2\text{)} \\
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & \lambda a.\,\mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![m_1]\!]_{\mathrm{c}} \\
 & & \quad\vdots \\
 & & \quad \mathit{else}\ \mathit{if}\ a =_A \ell_n\ \mathit{then}\ [\![m_n]\!]_{\mathrm{c}} \\
 & & \quad \mathit{else}\ \star \\
 & & \text{(where } a \text{ does not appear free in } m_i\text{)} \\
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & \mathit{let}\ x_1 = [\![m_1]\!]_{\mathrm{r}}\ \mathit{in} \\
 & & \vdots \\
 & & \mathit{let}\ x_n = [\![m_n]\!]_{\mathrm{r}}\ \mathit{in} \\
 & & \lambda a.\,\mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
 & & \quad\vdots \\
 & & \quad \mathit{else}\ \mathit{if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
 & & \quad \mathit{else}\ \star \\
 & & \text{(where } x_i \text{ does not appear free in } m_i\text{)} \\
[\![\pi_\ell(m)]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{c}}\ell \\
[\![\pi_\ell(m)]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{r}}\ell \\
[\![m : \sigma]\!]_{\mathrm{c}} & \stackrel{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{c}} \\
[\![m : \sigma]\!]_{\mathrm{r}} & \stackrel{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{r}}
\end{array}
$$

Figure 3.6: Embedding Modules

Proposition 3.2 *If $\vdash_\nu t$ in T and $t \mapsto^* t'$ then $\vdash_\nu t'$ in T.*

Proof

Direct from Corollary 4.4.

Corollary 3.3 (Type Preservation) *If $\vdash_K e : \tau$ and $[\![e]\!] \mapsto^* t$ then $\vdash_\nu t$ in $[\![\tau]\!]$.*

Another consequence of the soundness theorem is that the phase distinction [16, 46] is respected in λ^K: all type expressions converge and therefore types may be computed in a compile-time phase. This is expressed by Corollary 3.4:

Corollary 3.4 (Phase Distinction) *If $\vdash_K c : \kappa$ then there exists canonical t such that $[\![c]\!] \mapsto^* t$.*

Proof

For any well-formed λ^K kind κ, the embedded kind $[\![\kappa]\!]$ can easily be shown to be a total type. (Intuitively, every type is total unless it is constructed using the partial type constructor, which is not used in the embedding of kinds.) The conclusion follows directly.

3.5 Prospects for Extension

In order to embed λ^K into type theory, I abandoned impredicativity. This was because the standard semantics for Nuprl (discussed in Section 4.4) does not support it. However, Mendler [69] has developed a semantic model of Nuprl enhanced with recursive types and some impredicative polymorphism. In that type theory, it is entirely straightforward to handle second-order impredicativity. I have not used Mendler's type theory in this thesis because its semantics is very complicated, and because it is not clear how easily it can be extended to support partial types. An alternative type theory to be explored is the Calculus of Constructions [29, 28], which also supplies impredicative features and could likely support the semantics discussed in this chapter.

Another important avenue for future work is to extend the semantics in this chapter to explain stateful computation. One promising device for doing this is to encode stateful computations as monads [82, 60], but this raises two difficulties. In order to encode references in monads, all expressions that may side-effect the store must take the store as an argument. The problem is how to assign a type to the store. Since side-effecting functions may be in the store themselves, the store must be typed using a recursive type, and since side-effecting expressions take the store as an argument, that recursive type will include *negative* occurrences of the variable of recursion. Mendler's type theory may express recursive types with only positive occurrences, but to allow negative occurrences is an open problem.[9]

Finally, a particularly compelling direction is to extend the semantics to account for objects, and that turns out to require only the same mechanisms discussed above. In a weak sense, this semantics can already support objects; the existential object encoding of Pierce and Turner [83] uses only constructs available within the type theory used here. Unfortunately, that encoding is not practical in a predicative system, because it involves quantification over the types of an object's hidden instance variables. That quantification results in objects always belonging to a universe one level higher than their underlying code, which prevents such object from being first-class.[10] However, in an impredicative type theory, Pierce and Turner's object encoding can be used quite satisfactorily. In a type theory that additionally supplies recursive types (with negative occurrences), a variety of other object encodings become possible as well [15, 13, 3, 31, 14]. Alternatively, the object encoding of Hickey [50] works entirely within the existing Nuprl type theory and shows promise of being extendable to a practical object system.

3.6 Conclusions

I have shown how to give a type-theoretic semantics to an expressive programming calculus that supports modular and object-oriented features. This semantics makes it possible to use formal type-theoretic reasoning about programs and programming languages without informal embeddings and without sacrificing core expressiveness of the programming language.

Formal reasoning aside, embedding programming languages into type theory allows a researcher to bring the full power of type theory to bear on a programming problem. For example, in Crary [30] I use a type-theoretic interpretation to expose the relation of power kinds to a nonconstructive set type. Adjusting this interpretation to make the power kind constructive

[9] See Birkedal and Harper [11] for a promising approach that may lead to a solution of this problem.

[10] However, in some contexts it is not necessary for objects to be first class. For example, Jackson [57] independently used an encoding essentially the same as Pierce and Turner's to implement computational abstract algebra. In that context, algebras were rarely intermingled with the elements of an algebra, and when they were, an increase in the universe level was acceptable.

results in the proof-passing technique used to implement higher-order coercive subtyping in KML.

Furthermore, the simplicity of the semantics makes it attractive to use as a mathematical model similar in spirit, if not in detail, to the Scott-Strachey program [90]. This semantics works out so neatly because type theory provides built-in structure well-suited for analysis of programming. Most importantly, type theory provides structured data and an intrinsic notion of computation. Non-type-theoretic models of type theory can expose the "scaffolding" when one desires the details of how that structure may be implemented (Section 4.4).

As a theory of structured data and computation, type theory is itself a very expressive programming language. Practical programming languages are less expressive, but offer properties that foundational type theory does not, such as decidable type checking. I suggest that it is profitable to take type theory as a foundation for programming, and to view practical programming languages as *tractable approximations* of type theory. The semantics in this chapter illustrates how to formalize these approximations. This view not only helps to *explain* programming languages and their features, as I have done here, but also provides a greater insight into how we can bring more of the expressiveness of type theory into programming languages.

Chapter 4

Foundational Type Theory

In this chapter I present in detail the foundational type theory of Nuprl. Broadly speaking, the intent of the Nuprl type theory is to provide a formal foundation for mathematics that is intrinsically computational. Like the simpler type theories that provide the superstructure for various programming languages (such as λ^K), Nuprl is a theory of structured data and computation. The essential difference between Nuprl (and other foundational type theories [68, 29, 42]) and the simpler type systems of ordinary programming languages is the expressiveness of the type system. Foundational type theories provide type structure rich enough to encode (almost) arbitrary logical and mathematical propositions.

This logical character of the Nuprl type system is not immediately evident at a first inspection. Instead, the constructs that make up Nuprl are those of a fairly simple programming language: data such as integers, functions, and pairs, and types for classifying such data. Those computational constructs are interpreted as logical statements using the *propositions-as-types* correspondence discussed in Section 4.1.3. This conflating of logic and data is one of the interesting aspects of type theory, and is in contrast to most other mathematical foundations (such as set theory), which draw sharp distinctions between objects and logical sentences.

4.1 Nuprl

The standard Nuprl type theory is presented in Constable *et al.* [19]. I present here a variant of the Nuprl type theory with a number of innovations. Chief among those innovations are the partial type mechanisms added in order to make it possible to reason about potentially nonterminating computations. In Section 4.3 I discuss some of the design decisions for my version of Nuprl.

4.1.1 Basic Types and Operators

At the core of the type theory are the basic data types summarized in Figure 4.1. Each of these data types have a collection of introduction operators, and a collection of analysis (or elimination) operators. Nuprl does not have distinct syntactic classes for types and terms; instead, types are ordinary terms that may be computed using the same operations for computing data. For example, if A and B are types then so is *if b then A else B*. The basic types of Nuprl are the following:

- *Atom*: The collection of introduction forms for *Atom* is the set of string literals. The sole elimination form for *Atom* is the equality test $a =_A a'$, which is equivalent to *true* when a

	Type Formation	Introduction	Elimination
atoms	$Atom$	string literals	equality test ($=_A$)
integers	$\mathbb{Z}$	$\ldots, -1, 0, 1, 2, \ldots$	assorted operations $\oplus$
booleans	$\mathbb{B}$	$true, false$	$if\ e\ then\ e_1\ else\ e_2$
disjoint union	$T_1 + T_2$	$inj_1(e)$ $inj_2(e)$	$case(e, x_1.e_1, x_2.e_2)$
product space	$\Sigma x{:}T_1.T_2$	$\langle e_1, e_2 \rangle$	$\pi_1(e)$ $\pi_2(e)$
function space	$\Pi x{:}T_1.T_2$	$\lambda x.e$	$e_1 e_2$
very-dependent function	$\{f \mid x{:}T_1 \to T_2\}$	$\lambda x.e$	$e_1 e_2$
unit	$Unit$	$\star$	
void	$Void$		
universe i	$\mathbb{U}_i$ (for $i \geq 1$)	type formation operators	

Figure 4.1: Basic Nuprl Types

and a' are equal atoms, and is equivalent to *false* when a and a' are unequal atoms. (Note that $Atom$ has considerably less structure than a type of strings; the essential properties of $Atom$ are only a countably infinite set of introduction forms and an equality test.)

- *Integer* ($\mathbb{Z}$): The collection of introduction forms for $\mathbb{Z}$ is the set of integer literals. The elimination forms for integers are boolean equality and inequality tests ($=_Z$ and $\leq_Z$), and a collection of binary operations (ranged over by $\oplus$), such as plus, times, etc. For convenience, these operators are taken to be defined on all integer values and return zero in exceptional cases such as division by zero.

- *Boolean* ($\mathbb{B}$): The introduction forms for $\mathbb{B}$ are *true* and *false*. Booleans are eliminated by the branching construct *if* b *then* e_1 *else* e_2. (For simplicity, booleans will eventually be defined in terms of the unit and disjoint union types.)

- *Disjoint Union* ($A + B$): The introduction forms of $A + B$ are $inj_1(a)$ (when a is a term belonging to A) and $inj(b)$ (when b is a term belonging to B). Disjoint unions are eliminated by the case analysis construct $case(e, x_1.e_1, x_2.e_2)$.

- *Dependent Product* ($\Sigma x{:}A.B$): The introduction forms of $\Sigma x{:}A.B$ are $\langle a, b \rangle$, when a belongs to A and b belongs to $B[a/x]$. Disjoint unions are eliminated by the projection constructs $\pi_1(e)$ and $\pi_2(e)$. As usual, when x does not appear free in B, I write $\Sigma x{:}A.B$ as the non-dependent product $A \times B$. (This type is also known in the literature as a dependent sum, general sum or strong sum. I choose the term dependent product in order to emphasize the connection to the non-dependent product.)

- *Dependent Function* ($\Pi x{:}A.B$): The introduction form of $\Pi x{:}A.B$ is $\lambda x.b$, when $b[a/x]$ belongs to $B[a/x]$ for every a belonging to A. Functions are eliminated by the application construct $e_1 e_2$. As usual, when x does not appear free in B, I write $\Pi x{:}A.B$ as the non-dependent function type $A \to B$.

- *Very Dependent Function* ($\{f \mid x{:}A \to B\}$): The very dependent function type (due to Hickey [49]) has the same introduction form ($\lambda x.b$) as the ordinary dependent function type, but allows the result type B to depend not only on the argument x but on the function itself. Thus, $\lambda x.b$ belongs to $\{f \,|\, x{:}A \rightarrow B\}$ when $b[a/x]$ belongs to $B[\lambda x.b, a/f, x]$ for every a belonging to A.

To avoid circularity, in order for a very dependent function type $\{f \mid x{:}A.B\}$ to be well-formed, we require that B only use f by applying it to elements of A that are less than x with respect to some well-founded order. As discussed in Section 3.3.3, the very dependent function type is sufficient to encode many of the other basic types discussed here, but it is convenient (especially in Chapter 5) to retain those other types as primitive.

- *Unit*: The type *Unit* has the introduction form $\star$ and no elimination forms. (For simplicity, *Unit* will eventually be defined in terms of the equality type.)

- *Void*: The type *Void* is empty and therefore has no introduction forms.

- *Universe* ($\mathbb{U}_i$): Each type expression is Nuprl is also an ordinary term, and is a member of a universe type. The universe $\mathbb{U}_1$ is the type containing all *small* types: those that can be built without using a universe. The universe $\mathbb{U}_{i+1}$ then contains all types that can be built using no universe higher than $\mathbb{U}_i$. In particular, no universe is a member of itself; to include a single type that contained all types (including itself) would make the theory inconsistent [36, 27, 70, 52]. The introduction forms for a universe are the type formation operators. No elimination form for universes is provided, but this is only for simplicity; type analysis operations could easily be added, following Constable and Zlatin [20, 26].

4.1.2 Equality and Well-formedness

Central to the type theory are two equality relations, one for equality of types and another for equality of terms relative to a type. The type equality relation, written $T_1 = T_2$, states that T_1 and T_2 are each well-formed types and are equal to each other. The term equality relation, written $e_1 = e_2 \in T$, states that T is a type and that e_1 and e_2 are members of T and are equal when considered as members of T. A specification of these two relations appears in Figure 4.5.

The reflexive forms of these relations are particularly important. Note that $T = T$ exactly when T is a well-formed type, and that $e = e \in T$ exactly when e is a member of T. These reflexive cases are abbreviated as T type and $e \in T$, respectively.

For example, $\Pi x{:}A.B = \Pi x{:}A'.B'$ if and only if $A = A'$ and $B[a/x] = B'[a'/x]$ whenever $a = a' \in A$; and $\lambda x.b = \lambda x.b' \in \Pi x{:}A.B$ if and only if $\Pi x{:}A.B$ type and $b[a/x] = b'[a'/x] \in B[a/x]$ whenever $a = a' \in A$.

We will occasionally have need of a special degenerate type to serve as a supertype of all types:

- *Top*: For any terms e_1 and e_2, $e_1 = e_2 \in \mathit{Top}$. (For simplicity, *Top* will eventually be defined in terms of the *every* type.)

Notation 4.1 *I write* $\forall a \in A. P$ *to mean* $\forall a. a \in A \Rightarrow P$ *and* $\forall a = a' \in A. P$ *to mean* $\forall a, a'. a = a' \in A \Rightarrow P$.

4.1.3 Propositions as Types

In order to use type theory for mathematics, we need a way to make logical statements (propositions) within the formal theory. This is done using the *propositions-as-types* correspondence, also known as the Curry-Howard isomorphism [51]. In this correspondence, each logical sentence is interpreted as a type, with the understanding that that the sentence is true exactly when the corresponding type is inhabited by some term. That inhabitant is referred to as the witness of the proposition.

For example, the proposition *False* is interpreted as the uninhabited type *Void*. The other logical connectives are interpreted as follows:

$$\begin{array}{rcl}
\mathit{True} & \stackrel{\text{def}}{=} & \mathit{Unit} \\
P \wedge Q & \stackrel{\text{def}}{=} & P \times Q \\
P \vee Q & \stackrel{\text{def}}{=} & P + Q \\
P \Rightarrow Q & \stackrel{\text{def}}{=} & P \rightarrow Q \\
P \Leftrightarrow Q & \stackrel{\text{def}}{=} & (P \rightarrow Q) \times (Q \rightarrow P) \\
\neg P & \stackrel{\text{def}}{=} & P \rightarrow \mathit{Void} \\
\forall x{:}T.P & \stackrel{\text{def}}{=} & \Pi x{:}T.P \\
\exists x{:}T.P & \stackrel{\text{def}}{=} & \Sigma x{:}T.P
\end{array}$$

Higher-order logic may be interpreted by this device as well. We may state a proposition quantified over all (small) predicates as $\forall P{:}\mathbb{P}_1.Q$. This is interpreted in the type theory by defining the type of propositions $\mathbb{P}_i$ to be the universe $\mathbb{U}_i$.

4.1.4 Equality, Subtyping and Inequality

Although the preceding is sufficient to encode propositional and higher-order predicate logic, we are short on atomic propositions. For example, we would like to be able to speak about equality within the logic, but none of the types discussed so far is able to express such a proposition. To handle this sort of atomic assertion, we add new types that are defined to be inhabited by the trivial term $\star$ when the proposition they represent is true, and empty otherwise:

- *Equality* ($e_1 = e_2\ \mathit{in}\ T$): The type $e_1 = e_2\ \mathit{in}\ T$ is well-formed when $e_1 \in T$ and $e_2 \in T$. If $e_1 = e_2 \in T$ then the type $e_1 = e_2\ \mathit{in}\ T$ has the introduction form $\star$, if not then the type is empty (and is interpreted as a false proposition). Using the equality type, we may also define a membership type (denoted $e\ \mathit{in}\ T$) as $e = e\ \mathit{in}\ T$.

- *Subtyping* ($T_1 \preceq T_2$): The type $T_1 \preceq T_2$ is well-formed when T_1 and T_2 are well formed types. If T_1 is a subtype of T_2 (*i.e.*, $e = e' \in T_2$ whenever $e = e' \in T_1$) then $T_1 \preceq T_2$ has the introduction form $\star$, otherwise it is empty (and is interpreted as a false proposition).

- *Inequality* ($n_1 \leq n_2$): The type $n_1 \leq n_2$ is well-formed when $n_1 \in \mathbb{Z}$ and $n_2 \in \mathbb{Z}$ and is inhabited (by $\star$) when the integer value of is n_1 is less than or equal to that of n_2.

Be careful to note the distinction between $e_1 = e_2\ \mathit{in}\ T$, a type (and a proposition) within the formal type theory, and $e_1 = e_2 \in T$, a metatheoretical statement outside the theory.

Also note that subtyping is defined as type inclusion: if $T_1 \preceq T_2$ then the members of T_1 are actual members of T_2. An alternative definition would be that $T_1 \preceq T_2$ when exists a coercion from T_1 to T_2 [12]. This would provide an apparently more general notion of subtyping, since the identity function could always serve as a coercion from a type to another type that includes it. In its greatest generality, coercive subtyping may be represented by the proposition $T_1 \Rightarrow T_2$, which states that there exists a function from T_1 to T_2. Alternatively, if the coercion interpretation is restricted to use a predetermined collection of coercions, then the two notions of subtyping may be reconciled in a common type system [30]. This is done by building a new type system in which a type T_2 contains all the members of every type T_1 such that T_1 is coercible to T_2. Then the equalities on such types are constructed so that coercion results are deemed equal to the corresponding coercion arguments, and thus coercion functions may formally be viewed as identity functions.

Negatability Consider the proposed proposition $3\ \mathit{in}\ \mathit{Atom}$. Certainly it is not true, but it is not false either: Since the well-formedness criterion for the equality type requires its equands to belong to the stated type, $3\ \mathit{in}\ \mathit{Atom}$ is ill-formed and not a type at all. This seemingly innocuous phenomenon has considerable consequences; it means that the membership proposition is not *negatable.*

To understand why, consider the proposition $e\ \mathit{in}\ T$. If (metatheoretically speaking) $e \in T$ then $e\ \mathit{in}\ T$ is well-formed and true (*i.e.,* inhabited). On the other hand, if $e \notin T$ then $e\ \mathit{in}\ T$ is ill-formed and unusable. This does not mean that membership proposition is useless; on the contrary, it is quite often essential. However, the membership proposition *is* useless as the antecedent of an implication. As a simple example, the negation $\neg(e\ \mathit{in}\ T)$ is either false or ill-formed, but never true and hence never useful. More generally, the implication $e \in T \Rightarrow P$ is no easier to prove than P, because the well-formedness of the implication cannot be shown without proving the antecedent, rendering the antecedent useless.

The non-negatability phenomenon is not limited to the membership proposition. Consider the proposition $\forall x{:}T_1.\, x\ \mathit{in}\ T_2$. This proposition could be used to define the subtyping relation, but like the membership proposition, it is well-formed only when true. That is why subtyping is made a built-in type, because we would like to be able to prove theorems like $A \preceq B \Rightarrow \mathbb{Z} \rightarrow A \preceq \mathbb{Z} \rightarrow B$ and $\neg(\mathbb{U}_2 \preceq \mathbb{U}_1)$. In Section 4.1.8 we will see a few more examples of primitive types included to provide negatability.

4.1.5 Constructivity

The interpretation of propositions as types is convenient, but it also restricts the proof techniques that may be used to prove such propositions. Consider the proposition $XM = \forall P{:}\mathbb{P}_1.\, P \vee \neg P$. In classical logic, this proposition is certainly true: every proposition is either true or false (alternatively, every type is either inhabited or uninhabited). However, for XM to be valid in Nuprl, there must exist a function f that computes a member of $P + \neg P$ for every (small) proposition P. This is certainly impossible, since P may express an undecidable proposition.

The Nuprl logic restricts proofs to using only principles of reasoning that are constructive (or intuitionistic [33]). This means that any proof of the existence of an object must show how to construct it, and any proof of a disjunction must show how to determine which case is true. In practice this means that proofs may not use the principle of the excluded middle (proposition XM above) or proof by contradiction. (Contradiction may, however, be used to prove theorems of the form $\neg\neg P$.)

The dividend of this restriction is that proofs may be interpreted as programs. Suppose we prove $\forall x{:}\mathbb{Z}.\, \exists y{:}\mathbb{Z}.\, P(x, y)$ constructively. In so doing, we have constructed a function inhabiting the corresponding type $\Pi x{:}\mathbb{Z}.\, \Sigma y{:}\mathbb{Z}.\, P(x, y)$; this function is the *computational content* of the proof. The function may be applied to any integer x to compute the desired integer y and a proof of $P(x, y)$. More generally, whenever we prove the existence of objects, we may extract the computational content from that proof: a program to compute those objects. I discuss this process of extracting computational content from proofs further in Section 4.2.2. In Section 4.4.3 I discussed the controlled reintroduction of classical reasoning principles.

Propositions versus Booleans It is important to be clear on the distinction between the type $\mathbb{P}_i$ of propositions and the type $\mathbb{B}$ of booleans. Propositions are types that are inhabited when their intended meaning is true and are empty otherwise. Booleans are terms that compute to the value *true* when their intended meaning is true and to *false* otherwise. Decidable propositions may also be stated as boolean expressions; for example *true*, *false*, $a =_A a'$, $n =_Z n'$ and $n \leq_Z n'$ are boolean forms of the propositions *True*, *False*, $a = a'$ *in Atom*, $n = n'$ *in* $\mathbb{Z}$ and $n \leq n'$. Undecidable propositions can never be stated as boolean expressions because a computation that resulted in such a boolean would solve the undecidable problem.

4.1.6 Constructing New Types using Logic

We may also use logical predicates to construct variants of other types, using two special type constructors (due to Constable [21]):

- The *set type* $\{x{:}T \mid P\}$ is the subtype of T that contains all $t \in T$ such that $P[t/x]$ is true (*i.e.*, inhabited).
- The *quotient type* $xy{:}T//E[x,y]$ (when $E[-,-]$ is an equivalence relation on T) is the supertype of T that coarsens the equality on T as follows: $t_1 = t_2 \in xy{:}T//E$ if and only if $t_1, t_2 \in T$ and $E[t_1, t_2/x, y]$ is true (*i.e.*, inhabited).

We may use set and quotient types to define a number of important variants on existing types, such as the natural numbers, $\mathbb{N} \stackrel{\text{def}}{=} \{n : \mathbb{Z} \mid 0 \leq n\}$, and the integers modulo k, $\mathbb{Z}_k \stackrel{\text{def}}{=} mn{:}\mathbb{Z}//((m-n) \bmod k = 0 \textit{ in } \mathbb{Z})$. A number of other derived forms appear in Figure 4.2.

An important point about set and quotient types is that they suppress computational content of their predicate. Given that $n \in \{x : \mathbb{Z} \mid P[x]\}$, it is certainly the case that $P[n]$ is inhabited by some term, but there is no way to gain access to that term. An an extreme example, consider the squashed proposition $\downarrow P \stackrel{\text{def}}{=} \{x : \textit{Unit} \mid P\}$. Metatheoretically it is easy to see that P is inhabited exactly when $\downarrow P$ is, and the implication $P \Rightarrow \downarrow P$ can easily be shown valid. However, the converse $\downarrow P \Rightarrow P$ is not valid, because it is not possible in general to produce a proof of a proposition simply from knowing it to be true. In order words, it is not generally possible to reconstruct suppressed computational content.

However, whenever the suppressed predicate is recursively enumerable, it is possible to reconstruct computational content. This covers many uses, including those few uses discussed above. On the other hand, it often is desirable to suppress computational content. For example, a proof of $\forall x{:}\mathbb{Z}.\, \exists y{:}\mathbb{Z}.\, P(x,y)$ computes both an integer y and a witness to $P(x,y)$, but it may be the case that so long as $P(x,y)$ is true, the actual witness is not interesting. In such cases, returning the witness is cluttersome at best, and computationally expense at worst. It would then be more appropriate to prove the similar proposition $\forall x{:}\mathbb{Z}.\{y : \mathbb{Z} \mid P(x,y)\}$.

4.1.7 Computation

Intrinsic to the type theory is an operational semantics that defines how Nuprl terms compute. This computation system is call-by-name (in contrast to λ^K) as is defined by a small-step evaluation relation, $t_1 \mapsto t_2$, and a set of canonical forms to which terms evaluate. The computation system is defined to be the restriction to closed terms of the evaluation step relation

$$
\begin{array}{rcll}
\mathit{Unit} & \stackrel{\text{def}}{=} & 0 = 0 \ \mathit{in}\ \mathbb{Z} & \\
\mathbb{B} & \stackrel{\text{def}}{=} & \mathit{Unit} + \mathit{Unit} & \\
\mathit{true} & \stackrel{\text{def}}{=} & \mathit{inj}_1(\star) & \\
\mathit{false} & \stackrel{\text{def}}{=} & \mathit{inj}_2(\star) & \\
\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2 & \stackrel{\text{def}}{=} & \mathit{case}(b, x.e_1, x.e_2) & \text{(where } x \text{ is not free in } e_1,\, e_2\text{)} \\
e\ \mathit{in}\ T & \stackrel{\text{def}}{=} & e = e\ \mathit{in}\ T & \\
T\ \text{type} & \stackrel{\text{def}}{=} & T \preceq T & \\
n < m & \stackrel{\text{def}}{=} & \neg(n \geq m) & \\
\mathbb{N} & \stackrel{\text{def}}{=} & \{n : \mathbb{Z} \mid 0 \leq n\} & \\
\mathbb{Z}^+ & \stackrel{\text{def}}{=} & \{n : \mathbb{Z} \mid 0 < n\} & \\
\mathbb{Z}_k & \stackrel{\text{def}}{=} & mn{:}\mathbb{Z}/\!/((m-n) \bmod k = 0\ \mathit{in}\ \mathbb{Z}) & \\
\mathbb{Q} & \stackrel{\text{def}}{=} & xy{:}(\mathbb{Z} \times \mathbb{Z}^+)/\!/(\pi_1(x) \times \pi_2(y) = \pi_1(y) \times \pi_2(x)\ \mathit{in}\ \mathbb{Z}) & \\
\downarrow\! P & \stackrel{\text{def}}{=} & \{x : \mathit{Unit} \mid P\} & \text{(where } x \text{ is not free in } P\text{)} \\
t\ \mathit{halts} & \stackrel{\text{def}}{=} & t\ \mathit{in}!\ \mathbb{E} & \\
\mathit{Top} & \stackrel{\text{def}}{=} & xy{:}\overline{\mathbb{E}}/\!/\,\mathit{Unit} &
\end{array}
$$

Figure 4.2: Derived Forms

given in Figure 4.3. Whether a term is canonical is governed by its outermost operator; the canonical forms of Nuprl are the type formation constructs and the introduction forms (*i.e.*, the middle column of Figure 4.1). Two important properties of evaluation are that evaluation is deterministic and canonical forms are terminal:

Proposition 4.2 *If $t \mapsto t_1$ and $t \mapsto t_2$ then $t_1 \equiv t_2$. Moreover, if t is canonical then $t \not\mapsto t'$ for any t'.*

If $t \mapsto^* t'$ and t' is canonical then we say that t converges (abbreviated $t\downarrow$) and t converges to t' (abbreviated $t \Downarrow t'$). Note that if $t \Downarrow t_1$ and $t \Downarrow t_2$ then $t_1 \equiv t_2$ and that if t is canonical then $t \Downarrow t$.

The computation system may be used to define a computational equivalence relation (often known as *applicative bisimulation*) written $t_1 \sim t_2$. Computational equivalence will be formally defined in Chapter 5 (Definition 5.3). Intuitively, two terms are computationally equivalent if they both converge to canonical forms with the same outermost operator and the corresponding subterms are computationally equivalent, or if they both fail to converge. It may be shown (Lemma 5.5, using Howe's method [53]) that computational equivalence includes beta-equivalence.

The equalities on types are constructed to observe computational equivalence;[1] that is, if $t \in T$ and $t \sim t'$ then $t = t' \in T$. This enables a powerful form of untyped reasoning called *direct computation.* When two terms are computationally equivalent (for example, a beta redex and contractum), they are indistinguishable by the type theory and may be freely exchanged for each other in any expression or proof:

Theorem 4.3 *If T* type *and $T \sim T'$ then $T = T'$. If $t \in T$ and $t \sim t'$ then $t = t' \in T$.*

[1] However, unlike the type system considered by Howe [53], equality in this type theory does not observe computational approximation (Section 5.2.1). This is because of the presence of partial types (Section 4.1.8). For example, $\perp$ approximates 1, but it is not the case that $\perp = 1\ \mathit{in}\ \overline{\mathbb{Z}}$.

$$\frac{e \mapsto_s e'}{case(e, x_1.e_1, x_2.e_2) \mapsto_s case(e', x_1.e_1, x_2.e_2)}$$

$$\frac{}{case(inj_1(e), x_1.e_1, x_2.e_2) \mapsto_s e_1[e/x_1]} \qquad \frac{}{case(inj_2(e), x_1.e_1, x_2.e_2) \mapsto_s e_2[e/x_2]}$$

$$\frac{e_1 \mapsto_s e_1'}{e_1 e_2 \mapsto_s e_1' e_2} \qquad \frac{}{(\lambda x.e_1)e_2 \mapsto_s e_1[e_2/x]} \qquad \frac{e \mapsto_s e'}{\pi_1(e) \mapsto_s \pi_1(e')} \qquad \frac{}{\pi_1(\langle e_1, e_2\rangle) \mapsto_s e_1}$$

$$\frac{e \mapsto_s e'}{\pi_2(e) \mapsto_s \pi_2(e')} \qquad \frac{}{\pi_2(\langle e_1, e_2\rangle) \mapsto_s e_2}$$

$$\frac{e_1 \mapsto_s e_1'}{e_1\ (=_A/=_Z/\leq_Z)\ e_2 \mapsto_s e_1'\ (=_A/=_Z/\leq_Z)\ e_2} \qquad \frac{e_1 \text{ is a string/integer literal} \quad e_2 \mapsto_s e_2'}{e_1\ (=_A/=_Z/\leq_Z)\ e_2 \mapsto_s e_1\ (=_A/=_Z/\leq_Z)\ e_2'}$$

$$\frac{e_1 \text{ and } e_2 \text{ are string/integer literals, } i = 1 \text{ iff } e_1 \equiv e_2}{e_1\ (=_A/=_Z)\ e_2 \mapsto_s inj_i(\star)} \qquad \frac{e_1 \text{ and } e_2 \text{ are integer literals, } i = 1 \text{ iff } e_1 \leq e_2}{e_1 \leq_Z e_1 \mapsto inj_i(\star)}$$

$$\frac{e_1 \mapsto_s e_1'}{let\ \ x = e_1\ \ in\ e_2 \mapsto_s let\ \ x = e_1'\ \ in\ e_2} \qquad \frac{e_1 \text{ is canonical}}{let\ \ x = e_1\ \ in\ e_2 \mapsto_s e_2[e_1/x]}$$

$$\frac{e_1 \mapsto_s e_1'}{e_1 \oplus e_2 \mapsto_s e_1' \oplus e_2} \qquad \frac{e1 \text{ is an integer literal} \quad e_2 \mapsto_s e_2'}{e_1 \oplus e_2 \mapsto_s e_1 \oplus e_2'} \qquad \frac{e1 \text{ and } e2 \text{ are integer literals} \quad e_1 \oplus e_2 = e}{e_1 \oplus e_2 \mapsto_s e}$$

$$\frac{}{fix(e) \mapsto_s e\ fix(e)}$$

Figure 4.3: Operational Semantics for Nuprl

This theorem is codified in the type theory by the direct computation rules, Rules 8 through 15 (Appendix B). Since computational equivalence is undecidable (and since we wish to be able to effectively determine whether an instance of a rule is valid), the direct computation rules approximate computational equivalence with the auxiliary judgement $\vdash_\nu t \sim t'$ (where t and t' are possibly open terms).

A simple application of Theorem 4.3 and direct computation is type preservation:

Corollary 4.4

- **(Semantic type preservation)** *If $t \in T$ and $t \mapsto t'$ then $t' \in T$.*
- **(Syntactic type preservation)** *If $H \vdash_\nu t$ in T and $t \mapsto t'$ then $H \vdash_\nu t'$ in T.*

Proof

If $t \mapsto t'$ then $t \sim t'$ (Proposition 5.4) and $\vdash_\nu t \sim t'$ (Rule 14). Semantic preservation follows directly by Theorem 4.3 and syntactic preservation follows by application of Rules 8 and 11.

4.1.8 Partial Types

Of particular interest in the computation system is the operator *fix*, which allows recursive computation and is evaluated by the rule $\mathit{fix}(f) \mapsto f(\mathit{fix}(f))$.[2] The primary use of *fix* is to define recursive functions. Unfortunately, in the standard Nuprl type theory, recursion is not as general tool as it is in most programming languages. All of the basic types of Nuprl are total (that is, they contain only convergent terms), but in general there is no guarantee that a recursive function converges. Thus, a recursive function (generally, a recursively computed object) cannot be given a type unless it can be shown to converge, and it cannot be used without giving it a type. However, it is not always convenient (or possible) to prove that recursive functions terminate, and furthermore, it is sometimes interesting to reason about functions that are known not to terminate for all inputs.

In order to be able to assign types to partial functions and nonconvergent objects, we add a new type constructor:

- *Partial type* ($\overline{T}$): The partial type $\overline{T}$ (due to Constable and Smith [24, 25, 92]) is like a lifted version of T: It contains all members of T, plus all nonconvergent terms. Terms are equal in $\overline{T}$ if they have the same convergence behavior, and if they are equal in T when they converge. That is, $t = t' \in \overline{T}$ if and only if $t{\downarrow} \Leftrightarrow t'{\downarrow}$ and $t{\downarrow} \Rightarrow t = t' \in T$.

With this type available, we may express the type of partial functions from A to B as $A \to \overline{B}$.

It will also prove convenient to have a type of all convergent terms:

- *Every* ($\mathbb{E}$): For any convergent terms e_1 and e_2, $e_1 = e_2 \in \mathbb{E}$.

In order to reason about termination in the logic, we add a new type to represent the atomic proposition of convergence using the same device that we used to add equality, subtyping and inequality types:

- *Convergence* (t *in*! T): The type t *in*! T is well-formed when $t \in \overline{T}$ and is inhabited (by $\star$) exactly when $t{\downarrow}$.

[2] The use of a *fix* operator greatly simplifies some of the results in this thesis (particularly the proof of Theorem 5.9), but it could in principle be eliminated and replaced with the Y combinator.

Note that $t\ in!\ T$ is an inhabited type exactly when $t\ in\ T$ is. A distinct convergence proposition is needed in order to provide negatability, which is essential for any nontrivial termination reasoning. Observe that $t\ in!\ T$ is well-formed whenever $t \in \overline{T}$, but $t \in T$ is well-formed only when it is true.

Constable and Smith [25, 92, 91] use an untyped convergence assertion in their type systems. In this type theory I instead use a typed convergence assertion (as in Constable and Smith [24]). However, an untyped convergence assertion can be defined as $t\ halts = t\ in!\ \mathbb{E}$. (This is the primary motivation for including the $\mathbb{E}$ type.) Unfortunately, as we will see in Section 4.3.1, in the presence of equality the untyped version is rarely any more useful than the typed version.

Partial Type Formation The convergence type raises a somewhat surprising issue about the formation of partial types. The type equality rule for convergence types states that $t\ in!\ T$ and $t'\ in!\ T$ are equal when $t = t' \in \overline{T}$, and equal types should always have the same membership. Therefore, $\overline{T}$ should never equate convergent and non-convergent terms. However, we desire that T be a subtype of $\overline{T}$ (*i.e.*, if $t = t' \in T$ then $t = t' \in \overline{T}$), so T should never equate convergent and non-convergent terms either. Since many terms (*Top* for example) do equate terms with different convergence behavior, we must add this as a restriction to the well-formedness of partial types.

To not equate convergent and non-convergent terms is an awkward condition to maintain, so instead my theory imposes the stronger condition that for $\overline{T}$ to be well-formed, T must be total. It follows trivially that a total type cannot equal convergent and non-convergent terms. The obvious definition of totality, $\forall x{:}T.\, x\ in!\ T$, turns out to be non-negatable (since it depends upon the well-formedness of $\overline{T}$) so I add totality as another primitive type:

- *Totality* (T total): The type T total is well-formed when T type and is inhabited (by $\star$) exactly when T contains only terms that converge.

The Fixpoint Principle With partial types in the type theory, recursive functions can be typed using the fixpoint principle:

$$e\ in\ \overline{T} \to \overline{T} \Rightarrow \mathit{fix}(e)\ in\ \overline{T}$$

The use of the fixpoint principle was illustrated in Section 3.3.2. However, the fixpoint principle is not valid for every type; it is only valid for types that are *admissible.* Admissibility is defined formally in Chapter 5 (Definition 5.13). Intuitively, a type is admissible if it contains a recursive term whenever it contains a cofinite set of approximations to that term.

This issue does not arise in most ordinary programming languages because most type systems are not rich enough to allow the expression of inadmissible types. Nuprl is expressive enough to construct inadmissible types, but nevertheless most types are still admissible. Chapter 5 discusses admissibility in detail, and presents a number of techniques for showing types to be admissible.

Admissibility is introduced into the logic as another atomic proposition:

- *Admissibility* (T admiss): The type T admiss is well-formed when T type and is inhabited (by $\star$) exactly when T is admissible.

This completes the set of primitive Nuprl types (except for four additional atomic propositions introduced in Chapter 5). The list is summarized in Figure 4.4.

void	$Void$		equality	$e_1 = e_2$ *in* A
atom	$Atom$		subtyping	$A \preceq B$
integer	$\mathbb{Z}$		inequality	$n_1 \leq n_2$
disjoint union	$A + B$		convergence	e *in!* A
product space	$\Sigma x{:}A.B$		totality	A total
function space	$\Pi x{:}A.B$		admissibility	A admiss
very-dependent function	$\{f \mid x{:}A \to B\}$		set	$\{x : A \mid P\}$
partial type	$\overline{A}$		quotient	$xy{:}A//P$
universe	$\mathbb{U}_i$		every	$\mathbb{E}$

Figure 4.4: Primitive Nuprl Types

Sequencing The final construct in the partial type theory is the sequencing construct $let\ x = e_1\ in\ e_2$. This expression evaluates e_1 to canonical form (say v) then evaluates $e_2[v/x]$. Hence, the sequence term diverges if either e_1 or e_2 does. Since e_2 is only evaluated if e_1 converges, the e_2 portion of the sequencing term may be typed under the assumption that x has a total type (the syntax for Nuprl sequents is formally defined in the next section):

$$\frac{H \vdash_\nu e_1\ in\ \overline{T} \quad H; x{:}T \vdash_\nu e_2\ in\ \overline{T'} \quad H \vdash_\nu \overline{T'}\ \text{type}}{H \vdash_\nu let\ x = e_1\ in\ e_2\ in\ \overline{T'}}$$

As an aside, if $\overline{T}$ is reinterpreted as a monad type $M(T)$, then this is precisely the typing rule for the *bind* operator in monad systems [74, 95, 94]. A monad system would also have a *unit* operator to coerce T to $M(T)$; in this theory that is taken care of by subsumption, since $T \preceq \overline{T}$.

One may extend the theory to a full type theory of monads by dispensing with $T \preceq \overline{T}$, adding an explicit unit operator and adding rules comprising the monadic equality laws. This would provide a powerful mechanism for abstractly managing effects in general [34], not just the nontermination effects managed by the partial types. I have not done so in this type theory, preferring instead to keep the type theory closely tied to its operational semantics, thereby providing as much structure as possible at the expense of abstraction (recall Section 3.2.1).

4.2 Sequents and Proof

The Nuprl inference system has one judgement form, written as the sequent:

$$x_1{:}T_1; \ldots; x_n{:}T_n \vdash_\nu C \triangleleft t$$

This may be read informally as "t is a member of C when $x_1, \ldots, x_n$ are members of $T_1, \ldots, T_n$." A more precise interpretation is discussed in Section 4.2.1 and a formal semantics is defined in Section 4.4.2. The bindings $(x_1{:}T_1; \ldots; x_n{:}T_n)$ to the left of the turnstile are called *hypotheses,* C is called the *conclusion* and t is called the *witness.*

The inference rules for deriving judgements of this form are listed in Appendix B. By way of example, Rule 21 for applying subtyping amidst the hypothesis list is:

$$\frac{H; x : T'; J \vdash_\nu C \triangleleft s \quad H \vdash_\nu T \preceq T'}{H; x : T; J \vdash_\nu C \triangleleft s}$$

Note that in this rule the second subgoal is a subtyping proposition, which is computationally uninteresting (it contains only terms that evaluate to $\star$). Since that proposition is computationally uninteresting, the statement of the rule suppresses the witness. In many rules the consequent also is computationally uninteresting and the witness is suppressed in such rules as well. In those rules, the witness should be taken to be $\star$. An example of such a rule is Rule 21 for applying subtyping in an equality assertion:

$$\frac{H \vdash_\nu t_1 = t_2 \ in \ T' \quad H \vdash_\nu T' \preceq T}{H \vdash_\nu t_1 = t_2 \ in \ T}$$

Implicitly, this stands for the rule:

$$\frac{H \vdash_\nu t_1 = t_2 \ in \ T' \lhd t \quad H \vdash_\nu T' \preceq T \lhd t'}{H \vdash_\nu t_1 = t_2 \ in \ T \lhd \star}$$

4.2.1 Functionality

A sequent $x_1{:}T_1; \ldots ; x_n{:}T_n \vdash_\nu C \lhd t$ asserts not only membership of the witness t in the conclusion type C, it also asserts that the conclusion and witness are *functional* over the hypothesis variables. That is, if $\vec{t} = t_1, \ldots, t_n$ and $\vec{t'} = t'_1, \ldots, t'_n$ are equal substitutions for $x_1{:}T_1, \ldots, x_n{:}T_n$ (in a manner that will be made precise in Section 4.4.2) then $C[\vec{t}/\vec{x}] = C[\vec{t'}/\vec{x}]$ and $t[\vec{t}/\vec{x}] = t[\vec{t'}/\vec{x}] \in C[\vec{t}/\vec{x}]$. To see why this interpretation is necessary, consider the rule:

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu B \lhd b}{H \vdash_\nu A \rightarrow B \lhd \lambda x.b}$$

The consequent states that $\lambda x.b$ is a function over A; that is, if $a = a' \in A$ then $b[a/x] = b[a'/x] \in B$. Under a weaker, membership only, interpretation of sequents, the second subgoal is sufficient to conclude that $b[a/x] \in B$ and $b[a'/x] \in B$, but cannot guarantee equality. A functional interpretation of sequents (where sequents are seen as functions from hypotheses to the conclusion) is necessary to validate the many rules of this form.[3]

4.2.2 Extract-Style Proof

In order to prove some proposition P, the propositions-as-types correspondence dictates that one prove a sequent of the form $\vdash_\nu P \lhd t$. However, it is typically the case that one is only interested in proving the truth (inhabitation) of a proposition, and is not concerned that witness be a particular term. Consequently, in the Nuprl proof development system [19, 56], a user is presented with a sequent that omits the "$\lhd t$" witness term, even in cases when conclusion is computationally interesting. The rules are then stated in a manner that makes it possible to extract the witness from the proof. This is called *extract-style* proof. For example, consider again the rule:

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu B \lhd b}{H \vdash_\nu A \rightarrow B \lhd \lambda x.b}$$

In extract-style proof, this rule is applied to a sequent of the form $H \vdash_{\nu e} P \Rightarrow Q$ to produce subgoals $H \vdash_{\nu e} P$ type and $H; x : P \vdash_{\nu e} Q$. From the proof of the second subgoal is extracted a witness b, possibly containing free occurrences of the variable x, and that witness is used to construct the extract, $\lambda x.b$, of the original sequent. In fact, since the variable x is only used

[3] In some alternative semantics for type theory (such as Howe's set theoretic semantics [54]), equality in a type may be collapsed to actual object equality, trivially making the membership and functionality interpretations the same. Extending the semantics of Howe to account for all the devices in this type theory is an open problem, but one that shows some promise of feasibility.

in the extract, it may be made invisible to the user as well, resulting in the simple subgoal $H; P \vdash_{\nu e} Q$.

For extract-style proof to be possible, all the rules in the system must be of the proper form. Specifically, the rules must construct extracts in a bottom-up manner, or, in other words, the applicability of a rule must not depend on the extract, since the extract cannot be known at the time the rule is applied. However, the phrasing of the Nuprl proof rules that I state in Appendix B, is devised in the interest of making the type theory as simple as possible, not supporting extract-style proof, so some rules are not of the proper form. To support extract-style proof, those rules must be replaced with extract-compatible rules. Fortunately, the necessary rules are all derivable from the existing rules, so it is possible to support extract-style proof without any additional metatheoretical justification.

The necessary condition for extract-compatibility is best understood by examining some rules that violate it and seeing how they may be rewritten to follow it:

Extract and Membership The function type inhabitation rule discussed above is not among the basic rules of the inference system. Rather it is derivable from them the equality rule on function types:

$$\dfrac{\dfrac{H \vdash_\nu A \text{ type} \quad \dfrac{H; x : A \vdash_\nu B \lhd b}{H; x : A \vdash_\nu b \textit{ in } B}(5)}{H \vdash_\nu \lambda x.b \textit{ in } A \to B}(60)}{H \vdash_\nu A \to B \lhd \lambda x.b}(4)$$

We gain considerable economy in the derivation rules by leaving out such inhabitation rules and allowing them to be derived from the others, but in an extract-style system they must be built-in instead. This is because Rule 5 is not appropriate for extract-style proof:

$$\frac{H \vdash_\nu C \lhd t}{H \vdash_\nu t \textit{ in } C}$$

For this rule to be applicable, the extract t of the subproof must be the same term t that is specified in the membership proposition in the consequent, and there is no guarantee that it will. Another way of looking at this problem is that the rule with witnesses stripped out is unsound; it does not follow from the inhabitation of C that any particular term t belongs to C:

$$\frac{H \vdash_{\nu e} C}{H \vdash_{\nu e} t \textit{ in } C}\ (\textit{wrong})$$

In an extract-style proof system, Rule 5 must be omitted and replaced by an inhabitation rule to mirror each basic equality rule (*e.g.,* 47, 48, 53, 60). However, note that the converse rule, Rule 4, does not pose problems for extract-style proof.

Hypothesis Elimination Another class of rules that impose a restriction on the extract term is the hypothesis elimination rules. The hypothesis elimination rule for products requires (Rule 56) that all free occurrences of x and y in the extract term must appear as the pair $\langle x, y\rangle$:

$$\frac{H; x : A; y : B; J[\langle x, y\rangle/z] \vdash_\nu C[\langle x, y\rangle/z] \lhd s[\langle x, y\rangle/z]}{H; z : \Sigma x{:}A.B; J \vdash_\nu C \lhd s}\ (x, y \notin J, C, s)$$

To make this rule suitable for extract-style proof, it must be rephrased as the derivable rule:

$$\frac{H; x : A; y : B; J[\langle x, y\rangle/z] \vdash_\nu C[\langle x, y\rangle/z] \lhd s}{H; z : \Sigma x{:}A.B; J \vdash_\nu C \lhd s[\pi_1(z), \pi_2(z)/x, y]}$$

Similarly, the hypothesis elimination rule for disjoint unions,

$$\frac{\begin{array}{c}H; y : A; J[inj_1(y)/x] \vdash_\nu C[inj_1(y)/x] \triangleleft s[inj_1(y)/x] \\ H; y : B; J[inj_2(y)/x] \vdash_\nu C[inj_2(y)/x] \triangleleft s[inj_2(y)/x]\end{array}}{H; x : A + B; J \vdash_\nu C \triangleleft s} \; (y \notin J, C, s)$$

must be rephrased as the derivable rule:

$$\frac{\begin{array}{c}H; y : A; J[inj_1(y)/x] \vdash_\nu C[inj_1(y)/x] \triangleleft s_1 \\ H; y : B; J[inj_2(y)/x] \vdash_\nu C[inj_2(y)/x] \triangleleft s_2\end{array}}{H; x : A + B; J \vdash_\nu C \triangleleft case(x, y.s_1, y.s_2)}$$

In contrast to these two rules, the hypothesis elimination rules for types with "uninteresting" memberships, such as the equality types (Rule 99), do not pose a problem:

$$\frac{H; x : (t = t' \ in \ T); J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H; x : (t = t' \ in \ T); J \vdash_\nu C \triangleleft s}$$

In these rules it is not a problem for $\star$ terms to propagate upwards, so such rules may be applied simply by assuming that x does not appear free in s.

Hidden Variables A special case of hypothesis elimination that poses particular difficulties is that of set and quotient types. The hypothesis elimination rule for set types (Rule 86) is:

$$\frac{H; x : A; y : B; J \vdash_\nu C \triangleleft s}{H; x : \{x : A \mid B\}; J \vdash_\nu C \triangleleft s} \; (y \notin J, C, s)$$

As before, y is prohibited from appearing free in the extract term s, but in this case the rule cannot be rephrased to make the problem disappear. The essential problem is that y represents computational content that is not available, so the extract cannot depend upon it in any way. To enforce this restriction, we must introduce a new mechanism: When the rule is applied, the variable y is marked as *hidden,* establishing the invariant that it may not appear free in the extract:

$$\frac{H; x : A; [y : B]; J \vdash_{\nu e} C}{H; x : \{x : A \mid B\}; J \vdash_{\nu e} C}$$

This invariant is preserved by the inference system, and disallows the application of any rule that would use that variable. However, when descending into subgoals that do not contribute to the extract all hiding annotations may be removed.

It is important to note that this hidden variable mechanism is only a practical syntactic device, and has no metatheoretical significance. The sole metatheoretical issue is addressed in Rule 86 by the side condition that y not appear free in the extract.

4.3 Issues in Partial Object Type Theory

The type theory presented here joins together the expressiveness of the standard Nuprl type theory, which does not support reasoning about partial objects (that is, nonconvergent terms), and the type theory of Smith [92, 91], which addresses partial objects but does not support equality. This theory also enhances Smith's theory by a considerably wider class of types that are admissible for fixpoint induction (discussed at length in Chapter 5). Here I discuss some issues arising in the design and application of a partial object type theory, particularly those resulting from the combination of equality and partial objects.

4.3.1 Partial Computation

Given two terms a and b, each of which belong to $\overline{\mathbb{Z}}$, it is intuitively clear that the sum $a+b$ should also belong to $\overline{\mathbb{Z}}$. However, the rules for integer expressions deal with convergent terms:

$$\frac{H \vdash_\nu a \ in \ \mathbb{Z} \quad H \vdash_\nu b \ in \ \mathbb{Z}}{H \vdash_\nu a + b \ in \ \mathbb{Z}}$$

An informal argument for the desired membership is easy: To show that $a + b \ in \ \overline{\mathbb{Z}}$, suppose that $(a+b) \ halts$ and show $a+b \ in \ \mathbb{Z}$. If $(a+b) \ halts$, then $a \ halts$ and $b \ halts$. Consequently $a \ in \ \mathbb{Z}$ and $b \ in \ \mathbb{Z}$ and therefore $a+b \ in \ \mathbb{Z}$ by the rule.

Unfortunately, formalizing this argument as a derivation runs into a problem; to show the desired membership we incur the additional obligation of showing that the supposition $(a+b) \ halts$ is well-formed, as in the derivation below, and the eventual subgoal $a+b \ in \ \overline{\mathbb{E}}$ is no easier to prove than the original goal $a+b \ in \ \overline{\mathbb{Z}}$.

$$\cfrac{\cfrac{\cfrac{H \vdash_\nu a + b \ in \ \overline{\mathbb{E}}}{H \vdash_\nu (a+b) \ halts \ \text{type}}\,(108)}{\vdots} \qquad}{} $$

$$\frac{H \vdash_\nu (a+b) \ halts \Leftrightarrow (a+b) \ halts \qquad H; x : (a+b) \ halts \vdash_\nu a+b \ in \ \mathbb{Z}}{H \vdash_\nu a+b \ in \ \overline{\mathbb{Z}}}\,(71)$$

This difficulty does not reveal a defect in the logic. The problem is that in the presence of equality, as discussed in Section 4.2.1, a sequent says much more than its naive reading; it asserts the functionality of the conclusion over equal substitutions for the hypotheses. The informal argument at the top is perfectly valid as a metatheoretical argument for closed terms, but it does not extend to sequents.

However, it certainly would be a defect in the logic if goals of this form were unprovable. This could be settled by fiat, by simply adding new partial typing rules for each operator, but we would rather avoid such a heavy-handed solution. I present a more palatable solution in the next section.

4.3.2 Computation in Active Positions

Consider the monad-style variation of $a+b$ obtained by first computing the summands to canonical form and them summing their values: $let \ x = a \ in \ let \ y = b \ in \ x + y$. This is easily shown to belong to $\overline{\mathbb{Z}}$:

$$\frac{H \vdash_\nu a \ in \ \overline{\mathbb{Z}} \qquad \cfrac{H; x : \mathbb{Z} \vdash_\nu b \ in \ \overline{\mathbb{Z}} \quad \cfrac{\cfrac{}{H; x : \mathbb{Z}; y : \mathbb{Z} \vdash_\nu x + y \ in \ \mathbb{Z}}\,(44,1)}{H; x : \mathbb{Z}; y : \mathbb{Z} \vdash_\nu x + y \ in \ \overline{\mathbb{Z}}}\,(70)}{H; x : \mathbb{Z} \vdash_\nu (let \ y = b \ in \ x + y) \ in \ \overline{\mathbb{Z}}}\,(81)}{H \vdash_\nu (let \ x = a \ in \ let \ y = b \ in \ x + y) \ in \ \overline{\mathbb{Z}}}\,(81)$$

The desired result then follows by direct computation if it can be shown that $\vdash_\nu a+b \sim (let \ x = a \ in \ let \ y = b \ in \ x + y)$. Of course, in general $t_2[t_1/x]$ is *not* equivalent to $let \ x = t_1 \ in \ t_2$, because the latter necessarily evaluates t_1 but the former might not, which would lead to different convergence behavior if t_1 is divergent. However, in this case the term $a+b$ *does* evaluate a and b before converging.

The general principle at work here is that x and y are both in *active* positions in the term $x+y$. An active position of a term is one that must be evaluated before the term can converge. Canonical terms have no active positions, since they have already converged. The active positions of non-canonical terms are defined as follows:

Definition 4.5 *A term e* appears actively *in t if:*

- $t \equiv e$*, or*
- $t \equiv case(t_1, x.t_2, x.t_3)$ *and e appears actively in t_1, or*
- $t \equiv t_1 t_2$ *and e appears actively in t_1, or*
- $t \equiv \pi_i(t')$ *and e appears actively in t', or*
- $t \equiv t_1 =_A t_2$ *or* $t \equiv t_1 =_Z t_2$ *or* $t \equiv t_1 \leq_Z t_2$ *and e appears actively in t_1 or t_2, or*
- $t \equiv let\ \ x = t_1\ in\ t_2$ *and e appears actively in t_1 or t_2, or*
- $t \equiv t_1 \oplus t_2$ *and e appears actively in t_1 or t_2, or*
- $t \equiv \mathit{fix}(t')$ *and e appears actively in t'*

Lemma 4.6 *If x appears actively in t then $t[e/x] \sim let\ \ x = e\ in\ t$.*

This lemma is put into practice in the type theory as Rule 15:

$$\frac{x \text{ appears actively in } t_2}{\vdash_\nu t_2[t_1/x] \sim let\ \ x = t_1\ in\ t_2}$$

Another application of this device is for computational inducement. Observe that evaluating $a + b$ induces the evaluation of a and b, so if $a + b$ converges then so do a and b. This can be proven using active positions:

$$\dfrac{\dfrac{H \vdash_\nu a + b\ in!\ \mathbb{Z} \quad \dfrac{}{\vdash_\nu (let\ \ x = a\ in\ x + b) \sim a + b}\,(15, 11)}{H \vdash_\nu (let\ \ x = a\ in\ x + b)\ in!\ \mathbb{Z}}\,(8) \qquad H \vdash_\nu a\ in\ \overline{\mathbb{Z}}}{H \vdash_\nu a\ in!\ \mathbb{Z}}\,(80)$$

4.3.3 Fixpoint Induction

The principal device for reasoning about programs in the LCF system [38] was fixpoint induction. Informally, fixpoint induction says that if an "admissible" predicate holds for divergent terms and it is preserved by f, then it holds for the fixpoint of f. In notation: if $P[\bot]$ and if $P[x] \Rightarrow P[f(x)]$ then $P[\mathit{fix}(f)]$.

Using the fixpoint principle in our type theory, we may derive a similar induction rule:

$$\frac{\begin{array}{l} H; x : \overline{T}; y : (x\ in!\ T \Rightarrow P) \vdash_\nu P \\ H; x : \overline{T}; y : P; z : (fx\ in!\ T) \vdash_\nu P[fx/x] \\ H \vdash_\nu f\ in\ \overline{T} \to \overline{T} \\ H; x : \overline{T} \vdash_\nu P \text{ type} \\ H \vdash_\nu T \text{ total} \\ H \vdash_\nu T \text{ admiss} \\ H \vdash_\nu P \text{ admiss}\,(x : T) \\ H; x : \overline{T}; y : \downarrow P \vdash_\nu P \triangleleft e \end{array}}{H \vdash_\nu P[\mathit{fix}(f)/x] \triangleleft e[\mathit{fix}(f), \star/x, y]}$$

This rule looks considerably more complicated than the informal statement, so let us examine it in detail. The first subgoal,

$$H; x : \overline{T}; y : (x \ in! \ T \Rightarrow P) \vdash_{\nu} P$$

corresponds to the base case, $P[\bot]$. If one accepts classical reasoning principles, in particular the proposition $(x \ in! \ T) \vee \neg(x \ in! \ T)$, then this goal may be proven from the simpler goal $H \vdash_{\nu} P[\bot/x]$. In Section 4.4.3, I discuss how classical reasoning principles may be justified semantically if one takes a classical view of the metatheory. However, it is usually the case that the given subgoal can be proven whenever $H \vdash_{\nu} P[\bot/x]$ can, so classical methods are usually unnecessary.

The second subgoal,

$$H; x : \overline{T}; y : P; z : (fx \ in! \ T) \vdash_{\nu} P[fx/x]$$

corresponds to the inductive case, $P[x] \Rightarrow P[f(x)]$. The z hypothesis indicates that we are not in the base case (*i.e.,* $f(x) \neq \bot$), and may be dropped, if desired, to bring the goal more in line with the informal statement.

The next three subgoals are basic well-formedness conditions. The sixth subgoal, $H \vdash_{\nu} T$ admiss, is also a well-formedness condition, which is needed to take the fixpoint of f. The seventh subgoal, $H \vdash_{\nu} P\,\mathrm{admiss}(x : T)$, is the requirement on the admissibility of the predicate. (More on this in Section 5.3.1.)

The final subgoal is the most curious:

$$H; x : \overline{T}; y : \downarrow P \vdash_{\nu} P \triangleleft e$$

This goal stems from the uniformity condition for admissibility of set types (Section 5.3.3), but we may give an informal explanation directly in the context of fixpoint induction. Suppose v_0 and g are the computational content of the base case and inductive case, respectively. For simplicity, assume that $v_0 \in P[\bot/x]$ and $g \in P[t/x] \to P[f(t)/x]$ (for all $t \in \overline{T}$). Then we may build a sequence of P inhabitants for each of the finite approximations of $\mathit{fix}(f)$: Let $v_{i+1} = g(v_i)$ so $v_0 \in P[\bot/x]$, $v_1 \in P[f(\bot)/x]$, $v_2 \in P[f(f(\bot))/x]$, etc. We need a limit term to inhabit the proposition $P[\mathit{fix}(f)/x]$. However, the sequence terms $v_0, v_1, v_2, \ldots$ are unrelated by computational approximation (Section 5.2.1) and do not in general approach any limit.

The final subgoal provides a term e that allows us to build a *uniform* sequence of inhabitants that will approach a limit. Since the sequence $v_0, v_1, v_2, \ldots$ inhabits the various P propositions, the sequence $e[\bot/x], e[f(\bot)/x], e[f(f(\bot))/x], \ldots$ also inhabits the various P propositions, and the latter is guaranteed to approach a limit.

Lemma 4.7 *The fixpoint induction rule is derivable.*

Proof

The full derivation of the rule is tedious, so I sketch the argument. Let $T' = \{x : T \mid P\}$. The key points are to show $\overline{T'} \preceq \{x : \overline{T} \mid P\}$ and to show that $f \ in \ \overline{T'} \to \overline{T'}$. From the latter, and since T' is admissible (Rule 143 and subgoals 6, 7 and 8), we may conclude that $\mathit{fix}(f) \ in \ \overline{T'}$. Thus, by the first point, $\mathit{fix}(f) \ in \ \{x : \overline{T} \mid P\}$. We may then use subgoal 8 to reconstruct the suppressed computational content of $P[\mathit{fix}(f)/x]$.

To show $\overline{T'} \preceq \{x : \overline{T} \mid P\}$ uses the base case. Suppose $x \ in \ \overline{T'}$. We wish to show that $x \ in \ \overline{T}$ and P is inhabited. The former is easy, since $T' \preceq T$. To show the latter, suppose x halts

($i.e.,$ x *in*! T). Then x *in* T', so P is inhabited. Hence x *in*! $T \Rightarrow P$. By the base case subgoal, P is inhabited.

To show f *in* $\overline{T'} \to \overline{T'}$ uses the inductive case. Suppose x *in* $\overline{T'}$. We wish to show that fx *in* $\overline{T'}$. Suppose fx halts ($i.e.,$ fx *in*! T). Then we wish to show that fx *in* T'; that is, that fx *in* T and $P[fx/x]$ is inhabited. The former follows trivially from fx *in*! T. To show the latter, observe that x *in* $\{x : \overline{T} \mid P\}$ (since $\overline{T'} \preceq \{x : \overline{T} \mid P\}$). Consequently P is inhabited. By the inductive case subgoal, $P[fx/x]$ is inhabited. □

Two somewhat similar results to Lemma 4.7 were proven by Smith [92, 91]. Smith's results differed in that one used classical reasoning principles and was based on the then-conjectural admissibility of set types, and the other required the witnesses of the base and inductive cases to have particular forms.

4.3.4 Computational Induction

Smith's type theory also provided another principle, *computational induction,* for reasoning about partial objects. The idea to computational induction is that when a term converges, it induces a well-founded order on its computation's intermediate values. Computational induction is generally useful for reasoning about programs since it corresponds to the intuitive mechanism of reasoning about recursive programs by preconditions and postconditions, but it is particularly useful for proving *partial* correctness of programs. Other induction principles are inadequate for such proofs; if one could find a well-founded order on the inputs of a recursive function, one could show total correctness instead.

Unfortunately, computation is an intrinsically *intensional* property; that is, it requires reasoning about the structure of terms as well as about their result value. In contrast, my type theory is *extensional;* terms with the same result value are indistinguishable. Consequently, it is trivially true that $P[t_2] \Rightarrow P[t_1]$ when $t_1 \mapsto t_2$, and one may not draw any inductive conclusions from the fact. Smith's type theory was able to support computational induction because it supported no notion of equality or functionality.

In order to support computational induction, we need to add support for intensional reasoning about terms. We may do this by adding to the type theory representations for all terms, and by adding rules for reasoning intensionally about those representations and for linking those representations back to the terms they represent. Mechanisms for accomplishing this are discussed in detail in Constable and Crary [23].

4.3.5 Other Contributions

The version of the Nuprl type theory presented in this chapter makes a few minor contributions other than those contributions relating to partial types:

- By adding a primitive subtyping assertion, subtyping becomes negatable (recall Section 4.1.4). This makes it possible to prove theorems that depend upon subtyping conditions. More importantly, however, it makes it possible to define the power kind as $\mathcal{P}_i(T) = \{S : \mathbb{U}_i \mid S \preceq T\}$ (recall Section 3.3.3). Without a negatable subtyping assertion, this set type is ill-formed.

- This type theory adds rules for a typehood assertion, T type, which may be defined either as $\lambda x.x$ *in* $T \to T$ or as $T \preceq T$. I use the latter definition, in order to share some rules with the subtyping assertion. Using a typehood assertion streamlines dealings with well-formedness subgoals, such as the first subgoal of:

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu b \textit{ in } B}{H \vdash_\nu \lambda x.b \textit{ in } \Pi x{:}A.B}$$

In the standard Nuprl theory, which does not use typehood assertions, the well-formedness subgoal is stated as membership in some universe: A *in* $\mathbb{U}_i$. However, to prove such a subgoal requires the determination of which universe $\mathbb{U}_i$ the type A belongs to. This is often inconvenient, and is sometimes impossible. For example, sometimes typehood can only be shown using an inhabitation rule (Rule 19):

$$\frac{H \vdash_\nu t = t' \textit{ in } T}{H \vdash_\nu T \text{ type}}$$

This rule could not be stated in the standard theory, using universe membership, because it would need to conclude that T was a member of some *unknown* universe $\mathbb{U}_i$, and there is no facility for existentially quantifying universe indices.

The use of typehood assertions is also useful in that it makes the type theory extensible to systems where some types do not belong to any universe. For example, suppose the computation system were augmented with a noncanonical operator $U(-)$ with the evaluation rule $U(n) \mapsto_s \mathbb{U}_n$ (for integer literals n). Then $\Sigma x{:}\mathbb{Z}.\, U(x)$ would be a valid type, but since $U(x)$ spans the universes, the type would be too large to belong to any single universe.

- The inclusion of an inhabitation-to-membership rule (Rule 5) makes it possible to derive the type inhabitation rules from the rules governing equality on types. This cuts the number of rules nearly in half. As discussed in Section 4.2.2, an extract-style proof development system must reintroduce those rules, but they may be reintroduced as derived rules without additional metatheoretic justification.

4.4 Semantics

The remaining task is to show that the type theory presented here is consistent. As usual, I do this by showing it sound with respect to a semantics. The development of the semantics is divided into two parts; I begin by giving a semantics for types and then extend that semantics to cover judgements as well. I give this semantics in the framework of partial equivalence relations. Semantics exist for related theories (standard Nuprl or Martin-Löf type theory [67, 68]) in the frameworks of set theory [54] and domain theory [87, 81, 80], among others [4, 10], and could probably be developed for my theory as well.

4.4.1 Type Semantics

The semantics of types is characterized by the relations $t_1 = t_2 \in T$ and $T_1 = T_2$ from Section 4.1.2. The first relation associates with each type T a partial equivalence relation,[4] $- = - \in T$, over the set of terms. That relation dictates which terms belong to the type and which are equal within that type. The second relation is another partial equivalence relation over terms that dictates which terms are types and which are equal types.

[4] A symmetric and transitive relation.

$$
\begin{array}{ll}
t \in T & \text{iff } t = t \in T \\
T \ \text{type} & \text{iff } T = T \\
T_1 = T_2 & \text{iff } \exists T_1', T_2'.\, (T_1 \Downarrow T_1') \wedge (T_2 \Downarrow T_2') \wedge (T_1' = T_2') \\
t_1 = t_2 \in T & \text{iff } \exists t_1', t_2', T'.\, (t_1 \Downarrow t_1') \wedge (t_2 \Downarrow t_2') \wedge (T \Downarrow T') \wedge (t_1' = t_2' \in T')
\end{array}
$$

$$
\begin{array}{ll}
\neg(t_1 = t_2 \in \mathit{Void}) & \\
a = a' \in \mathit{Atom}\ (a,\, a' \text{ atoms}) & \text{iff } a \equiv a' \\
n = n' \in \mathbb{Z}\ (n,\, n' \text{ integers}) & \text{iff } n \equiv n' \\
t = t' \in \mathbb{E} & \text{iff } t\!\downarrow \wedge\, t'\!\downarrow \\
\mathit{inj}_1(a) = \mathit{inj}_1(a') \in A + B & \text{iff } A + B \ \text{type} \wedge a = a' \in A \\
\mathit{inj}_2(b) = \mathit{inj}_2(b') \in A + B & \text{iff } A + B \ \text{type} \wedge b = b' \in B \\
\langle a, b\rangle = \langle a', b'\rangle \in \Sigma x{:}A.B & \text{iff } \Sigma x{:}A.B \ \text{type} \wedge (a = a' \in A) \wedge (b = b' \in B[a/x]) \\
\lambda x.b = \lambda x.b' \in \Pi x{:}A.B & \text{iff } \Pi x{:}A.B \ \text{type} \wedge \forall a = a' \in A.\, b[a/x] = b'[a'/x] \in B[a/x] \\
\lambda x.b = \lambda x.b' \in \{f \mid x{:}A \to B\} & \text{iff } \{f \mid x{:}A \to B\} \ \text{type} \wedge \forall a = a' \in A.\, b[a/x] = b'[a'/x] \in B[\lambda x.b, a/f, x] \\
t = t' \in \overline{T} & \text{iff } \overline{T} \ \text{type} \wedge (t\!\downarrow \Leftrightarrow t'\!\downarrow) \wedge (t\!\downarrow \Rightarrow t = t' \in T) \\
a = a' \in \{x : A \mid B\} & \text{iff } \{x : A \mid B\} \ \text{type} \wedge a = a' \in A \wedge \exists b.\, b \in B[a/x] \\
a = a' \in xy{:}A/\!/B & \text{iff } (xy{:}A/\!/B) \ \text{type} \wedge a \in A \wedge a' \in A \wedge \exists b.\, b \in B[a, a'/x, y] \\
\star \in (a = a' \ \mathit{in}\ A) & \text{iff } (a = a' \ \mathit{in}\ A) \ \text{type} \wedge (a = a' \in A) \\
\star \in (A \preceq B) & \text{iff } (A \preceq B) \ \text{type} \wedge \forall a = a' \in A.\, a = a' \in B \\
\star \in (t \leq t') & \text{iff } (t \leq t') \ \text{type} \wedge \exists n, n'.\, t \Downarrow n \wedge t' \Downarrow n' \wedge n \leq n' \\
\star \in (a \ \mathit{in!}\ A) & \text{iff } (a \ \mathit{in!}\ A) \ \text{type} \wedge a\!\downarrow \\
\star \in (A \ \text{total}) & \text{iff } (A \ \text{total}) \ \text{type} \wedge \forall t \in A.\, t\!\downarrow \\
\star \in (A \ \text{admiss}) & \text{iff } (A \ \text{admiss}) \ \text{type} \wedge \mathrm{Adm}(A)
\end{array}
$$

For $T \simeq T'$ iff $T = T'$, and for $T \simeq T'$ iff $T = T' \in \mathbb{U}_i$:

$$
\begin{array}{ll}
\mathit{Void} \simeq \mathit{Void} \qquad \mathit{Atom} \simeq \mathit{Atom} & \mathbb{Z} \simeq \mathbb{Z} \qquad \mathbb{E} \simeq \mathbb{E} \\
A + B \simeq A' + B' & \text{iff } A \simeq A' \wedge B \simeq B' \\
\Sigma x{:}A.B \simeq \Sigma x{:}A'.B' & \text{iff } A \simeq A' \wedge \forall a = a' \in A.\, B[a/x] \simeq B'[a'/x] \\
\Pi x{:}A.B \simeq \Pi x{:}A'.B' & \text{iff } A \simeq A' \wedge \forall a = a' \in A.\, B[a/x] \simeq B'[a'/x] \\
\{f \mid x{:}A \to B\} \simeq \{f \mid x{:}A' \to B'\} & \text{iff } A \simeq A' \wedge \\
& \quad \exists P, <.\, \forall a_1 = a_1' \in A.\, \forall a_2 = a_2' \in A.\, P[a_1, a_2/x, y] \simeq P[a_1', a_2'/x, y] \wedge \\
& \qquad\qquad \forall a, a'.\, a < a' \Leftrightarrow (\exists e.\, e \in P[a, a'/x, y]) \wedge < \text{ is well-founded} \wedge \\
& \qquad\qquad \forall a = a' \in A.\, \forall t = t' \in \{f \mid x{:}\{e : A \mid P[e, a/x, y]\} \to B\}. \\
& \qquad\qquad\quad B[t, a/f, x] \simeq B'[t', a'/f, x] \\
\overline{T} \simeq \overline{T'} & \text{iff } T \simeq T' \wedge \forall t \in T.\, t\!\downarrow \\
\{x : A \mid B\} \simeq \{x : A' \mid B'\} & \text{iff } A \simeq A' \wedge \forall a = a' \in A.\, B[a/x] \simeq B[a'/x] \wedge \\
& \quad \forall a = a' \in A.\, B'[a/x] \simeq B'[a'/x] \wedge \\
& \quad \exists t.\, t \in \Pi x{:}A.\, B \to B' \wedge \exists t.\, t \in \Pi x{:}A.\, B' \to B \\
(xy{:}A/\!/B) \simeq (xy{:}A'/\!/B') & \text{iff } A \simeq A' \wedge \\
& \quad \forall a_1 = a_1' \in A.\, \forall a_2 = a_2' \in A.\, B[a_1, a_2/x, y] \simeq B[a_1', a_2'/x, y] \wedge \\
& \quad \forall a_1 = a_1' \in A.\, \forall a_2 = a_2' \in A.\, B'[a_1, a_2/x, y] \simeq B'[a_1', a_2'/x, y] \wedge \\
& \quad \exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, B \to B' \wedge \exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, B' \to B \wedge \\
& \quad \exists t.\, t \in \Pi x{:}A.\, B[x/y] \wedge \exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, B \to B[y, x/x, y] \wedge \\
& \quad \exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, \Pi z{:}A.\, B \to B[y, z/x, y] \to B[z/y] \\
(a_1 = a_2 \ \mathit{in}\ A) \simeq & \\
\qquad (a_1' = a_2' \ \mathit{in}\ A') & \text{iff } A \simeq A' \wedge a_1 = a_1' \in A \wedge a_2 = a_2' \in A \\
(A \preceq B) \simeq (A' \preceq B') & \text{iff } A \simeq A' \wedge B \simeq B' \\
(t_1 \preceq t_2) \simeq (t_1' \preceq t_2') & \text{iff } t_1 = t_1' \in \mathbb{Z} \wedge t_2 = t_2' \in \mathbb{Z} \\
(a \ \mathit{in!}\ A) \simeq (a' \ \mathit{in!}\ A') & \text{iff } A \simeq A' \wedge a = a' \in \overline{A} \\
(A \ \text{total}) \simeq (A' \ \text{total}) & \text{iff } A \simeq A' \\
(A \ \text{admiss}) \simeq (A' \ \text{admiss}) & \text{iff } A \simeq A' \\
\mathbb{U}_i \ \text{type} & \\
\mathbb{U}_i \in \mathbb{U}_j & \text{iff } i < j
\end{array}
$$

Figure 4.5: Type Specifications

These relations are specified in Figure 4.5. Unfortunately, the specification contains negative occurrences of the relations being defined, so it cannot be taken as a valid inductive definition. However, examination reveals that each clause of the specification uses the relations only at types smaller than the type at issue in the clause. This suggests a way that we may rephrase the specification as a valid inductive definition. The technique I use is due to Allen [5]; my contribution here is its extension to a number of new types. An alternative but similar technique is that of Harper [40], who shows how the specification can be rephrased to define an explicit set theoretic relation as the least fixed point of a monotone operator.

The principal notion in the construction is that of a *type system*. A *candidate type system* τ is a relation $\tau T T' \phi$ between closed terms T and T' and binary relations ϕ over closed terms. The intended reading of $\tau T T' \phi$ is that T and T' are equal types and ϕ is their equality relation.

Definition 4.8 *A* type system *is a candidate type system τ that satisfies seven properties:*

- *τ is uniquely valued: if $\tau T T' \phi$ and $\tau T T' \phi'$ then ϕ is ϕ'*
- *τ is type symmetric: if $\tau T T' \phi$ then $\tau T' T \phi$*
- *τ is type transitive: if $\tau T_1 T_2 \phi$ and $\tau T_2 T_3 \phi$ then $\tau T_1 T_3 \phi$*
- *τ is type value respecting: if $\tau T T \phi$ and $T \mapsto T'$ then $\tau T T' \phi$*
- *τ is term symmetric: if $\tau T T' \phi$ then ϕ is symmetric*
- *τ is term transitive: if $\tau T T' \phi$ then ϕ is transitive*
- *τ is term value respecting: if $\tau T T' \phi$ and $t \, \phi \, t$ and $t \mapsto t'$ then $t \, \phi \, t'$*

The goal is to build a type system that captures the intended meanings of the types. I do this be defining a series of operators on candidate type systems. Note that each operator uses it argument τ only in positive positions, so they may be assembled into a valid inductive definition.

The first (and most important) step of the construction is to define, for each type constructor, an operator on candidate type systems. Given candidate type system τ, each operator is to return a type system that contains all types built using its type constructor from types in τ. For example, $\textsc{Prod}(\tau)$ will contain all product types built from types in τ. These operators define the meanings of the type constructors. Their definitions appear in Figures 4.6 through 4.8.

Next we define another operator $\textsc{Close}$ on candidate type systems. Given type system τ, $\textsc{Close}(\tau)$ contains all types built from types in τ using any type constructor other than universes. The intention is that the argument τ define all universes up to some $\mathbb{U}_i$, in which case $\textsc{Close}(\tau)$ will define all types built from those universes, that is, all members of $\mathbb{U}_{i+1}$. $\textsc{Close}(\tau)$ is defined to be the smallest candidate type system such that:

$$\begin{array}{l} \textsc{Close}(\tau)TT'\phi \ \ \textit{iff} \ \ \tau T T' \phi \vee (\bigvee_{\textsc{op} \in \textsc{Types}} \textsc{op}(\textsc{Close}(\tau))TT'\phi) \\ \text{where } \textsc{Types} = \left\{ \begin{array}{l} \textsc{Void}, \textsc{Atom}, \textsc{Int}, \textsc{Every}, \textsc{Union}, \\ \textsc{Prod}, \textsc{Fun}, \textsc{VFun}, \textsc{Bar}, \textsc{Eq}, \textsc{Sub}, \\ \textsc{Ineq}, \textsc{Conv}, \textsc{Total}, \textsc{Adm}, \textsc{Set}, \textsc{Quot} \end{array} \right\} \end{array}$$

Next we use mutual induction to define two series of candidate type systems, $\textsc{Univ}_i$ and $\textsc{Nuprl}_i$. $\textsc{Univ}_i$ defines all universes $\mathbb{U}_j$ where $j < i$, and $\textsc{Nuprl}_i$ defines all types built up from those

$$
\begin{array}{rll}
\textsc{Void}(-)TT'\phi & \textit{iff} & T \Downarrow \textit{Void} \wedge T' \Downarrow \textit{Void} \wedge \forall t, t'.\, \neg(t\ \phi\ t') \\
\textsc{Atom}(-)TT'\phi & \textit{iff} & T \Downarrow \textit{Atom} \wedge T' \Downarrow \textit{Atom} \wedge \forall t, t'.\, t\ \phi\ t' \Leftrightarrow (\exists a \in \{\text{atoms}\}.\, t \Downarrow a \wedge t' \Downarrow a) \\
\textsc{Int}(-)TT'\phi & \textit{iff} & T \Downarrow \mathbb{Z} \wedge T' \Downarrow \mathbb{Z} \wedge \forall t, t'.\, t\ \phi\ t' \Leftrightarrow (\exists n \in \{\text{integers}\}.\, t \Downarrow n \wedge t' \Downarrow n) \\
\textsc{Every}(-)TT'\phi & \textit{iff} & T \Downarrow \mathbb{E} \wedge T' \Downarrow \mathbb{E} \wedge \forall t, t'.\, t\ \phi\ t' \Leftrightarrow (t\!\downarrow \wedge\ t'\!\downarrow)
\end{array}
$$

$$
\begin{array}{rll}
\textsc{Union}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', \alpha, \beta. \\
 & & \quad T \Downarrow A + B \wedge T' \Downarrow A' + B' \wedge \tau AA'\alpha \wedge \tau BB'\beta \wedge \\
 & & \quad \forall t, t'.\, t\ \phi\ t' \Leftrightarrow \\
 & & \qquad (\exists a, a'.\, t \Downarrow \textit{inj}_1(a) \wedge t' \Downarrow \textit{inj}_1(a') \wedge a\ \alpha\ a') \vee \\
 & & \qquad (\exists b, b'.\, t \Downarrow \textit{inj}_2(b) \wedge t' \Downarrow \textit{inj}_2(b') \wedge b\ \alpha\ b') \\
\textsc{Prod}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', \alpha, \beta_{(-)}. \\
 & & \quad T \Downarrow \Sigma x{:}A.B \wedge T' \Downarrow \Sigma x{:}A'.B' \wedge \tau AA'\alpha \wedge \\
 & & \quad \forall a, a'.\, a\ \alpha\ a' \Rightarrow \tau(B[a/x])(B'[a'/x])\beta_a \wedge \\
 & & \quad \forall t, t'.\, t\ \phi\ t' \Leftrightarrow \\
 & & \qquad \exists a, b, a', b'.\, t \Downarrow \langle a, b\rangle \wedge t' \Downarrow \langle a', b'\rangle \wedge a\ \alpha\ a' \wedge b\ \beta_a\ b' \\
\textsc{Fun}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', \alpha, \beta_{(-)}. \\
 & & \quad T \Downarrow \Pi x{:}A.B \wedge T' \Downarrow \Pi x{:}A'.B' \wedge \tau AA'\alpha \wedge \\
 & & \quad \forall a, a'.\, a\ \alpha\ a' \Rightarrow \tau(B[a/x])(B'[a'/x])\beta_a \wedge \\
 & & \quad \forall t, t'.\, t\ \phi\ t' \Leftrightarrow \\
 & & \qquad \exists b, b'.\, t \Downarrow \lambda x.b \wedge t' \Downarrow \lambda x.b' \wedge \\
 & & \qquad\quad \forall a, a'.\, a\ \alpha\ a' \Rightarrow b[a/x]\ \beta_a\ b'[a'/x] \\
\textsc{VFun}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', P, <, \alpha, \gamma_{(-)}, \delta_{(-,-)}, \rho_{(-,-)}. \\
 & & \quad T \Downarrow \{f \mid x{:}A \to B\} \wedge T \Downarrow \{f \mid x{:}A' \to B'\} \wedge \tau AA'\alpha \wedge \\
 & & \quad \forall a_1, a_2, a_1', a_2'.\, a_1\ \alpha\ a_1' \Rightarrow a_2\ \alpha\ a_2' \Rightarrow \\
 & & \qquad \tau(P[a_1, a_2/x, y])(P[a_1', a_2'/x, y])\rho_{(a_1,a_2)} \wedge \\
 & & \quad \forall a, a'.\, a < a' \Leftrightarrow (\exists e.\, e\ \rho_{(a,a')}\ e) \wedge < \text{is well-founded} \wedge \\
 & & \quad \forall a, t, a', t'.\, a\ \alpha\ a' \Rightarrow t\ \gamma_a\ t' \Rightarrow \tau(B[t, a/f, x])(B'[t', a'/f, x])\delta_{(t,a)} \wedge \\
 & & \quad \forall e, t, t'.\, e\ \alpha\ e \Rightarrow \\
 & & \qquad t\ \gamma_e\ t' \Leftrightarrow \\
 & & \qquad\quad \exists b, b'.\, t \Downarrow \lambda x.b \wedge t' \Downarrow \lambda x.b' \wedge \\
 & & \qquad\qquad \forall a, a'.\, a\ \alpha\ a' \Rightarrow a < e \Rightarrow b[a/x]\ \delta_{(t,a)}\ b'[a'/x]) \wedge \\
 & & \quad \forall t, t'.\, t\ \phi\ t' \Leftrightarrow \\
 & & \qquad \exists b, b'.\, t \Downarrow \lambda x.b \wedge t' \Downarrow \lambda x.b' \wedge \\
 & & \qquad\quad \forall a, a'.\, a\ \alpha\ a' \Rightarrow b[a/x]\ \delta_{(t,a)}\ b'[a'/x] \\
\textsc{Bar}(\tau)TT'\phi & \textit{iff} & \exists A, A', \alpha. \\
 & & \quad T \Downarrow \overline{A} \wedge T' \Downarrow \overline{A'} \wedge \tau AA'\alpha \wedge \\
 & & \quad \forall a.\, a\ \alpha\ a \Rightarrow a\!\downarrow \wedge \\
 & & \quad \forall t, t'.\, t\ \phi\ t' \Leftrightarrow \\
 & & \qquad (t\!\downarrow \Leftrightarrow t'\!\downarrow) \wedge (t\!\downarrow \Rightarrow t\ \alpha\ t')
\end{array}
$$

Figure 4.6: Type Definitions (Data Types)

$$
\begin{array}{rll}
\textsc{Eq}(\tau)TT'\phi & \textit{iff} & \exists A, A', a_1, a_2, a_1', a_2', \alpha. \\
& & \quad T \Downarrow (a_1 = a_2 \ in \ A) \wedge T' \Downarrow (a_1' = a_2' \ in \ A') \wedge \\
& & \quad \tau A A' \alpha \wedge a_1 \ \alpha \ a_1' \wedge a_2 \ \alpha \ a_2' \wedge \\
& & \quad \forall t, t'. \, t \ \phi \ t' \Leftrightarrow (t \Downarrow \star \wedge t' \Downarrow \star \wedge a_1 \ \alpha \ a_2) \\
\textsc{Sub}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', \alpha, \beta. \\
& & \quad T \Downarrow (A \preceq B) \wedge T' \Downarrow (A' \preceq B') \wedge \tau A A' \alpha \wedge \tau B B' \beta \wedge \\
& & \quad \forall t, t'. \, t \ \phi \ t' \Leftrightarrow (t \Downarrow \star \wedge t' \Downarrow \star \wedge \forall a, a'. \, a \ \alpha \ a' \Rightarrow a \ \beta \ a') \\
\textsc{Ineq}(-)TT'\phi & \textit{iff} & \exists e_1, e_2, e_1', e_2'. \, \exists n_1, n_2 \in \{\text{integers}\}. \\
& & \quad T \Downarrow (e_1 \leq e_2) \wedge T' \Downarrow (e_1' \leq e_2') \wedge \\
& & \quad e_1 \Downarrow n_1 \wedge e_1' \Downarrow n_1 \wedge e_2 \Downarrow n_2 \wedge e_2' \Downarrow n_2' \wedge \\
& & \quad \forall t, t'. \, t \ \phi \ t' \Leftrightarrow (t \Downarrow \star \wedge t' \Downarrow \star \wedge n_1 \leq n_2) \\
\textsc{Conv}(\tau)TT'\phi & \textit{iff} & \exists A, A', a, a', \alpha. \\
& & \quad T \Downarrow (a \ in! \ A) \wedge T' \Downarrow (a' \ in! \ A') \wedge \tau A A' \alpha \wedge \\
& & \quad (\forall e. \, e \ \alpha \ e \Rightarrow e\downarrow) \wedge (a\downarrow \Leftrightarrow a'\downarrow) \wedge (a\downarrow \Rightarrow a \ \alpha \ a') \wedge \\
& & \quad \forall t, t'. \, t \ \phi \ t' \Leftrightarrow (t \Downarrow \star \wedge t' \Downarrow \star \wedge a\downarrow) \\
\textsc{Total}(\tau)TT'\phi & \textit{iff} & \exists A, A', \alpha. \\
& & \quad T \Downarrow (A \ \text{total}) \wedge T' \Downarrow (A' \ \text{total}) \wedge \tau A A' \alpha \wedge \\
& & \quad \forall t, t'. \, t \ \phi \ t' \Leftrightarrow (t \Downarrow \star \wedge t' \Downarrow \star \wedge \forall a. \, a \ \alpha \ a \Rightarrow a\downarrow) \\
\textsc{Adm}(\tau)TT'\phi & \textit{iff} & \exists A, A', \alpha. \\
& & \quad T \Downarrow (A \ \text{admiss}) \wedge T' \Downarrow (A' \ \text{admiss}) \wedge \tau A A' \alpha \wedge \\
& & \quad \forall t, t'. \, t \ \phi \ t' \Leftrightarrow \\
& & \qquad t \Downarrow \star \wedge t' \Downarrow \star \wedge \\
& & \qquad \forall f, e, e'. \, (\exists j. \forall k \geq j. \, e^{[k]_f} \ \alpha \ e'^{[k]_f}) \Rightarrow e^{[\omega]_f} \ \alpha \ e'^{[\omega]_f}
\end{array}
$$

Figure 4.7: Type Definitions (Assertions)

universes.

$$
\begin{array}{rll}
\textsc{Univ}_i TT'\phi & \textit{iff} & \exists j < i. \, T \Downarrow \mathbb{U}_j \wedge T' \Downarrow \mathbb{U}_j \wedge \\
& & \qquad \forall A, A'. \, A \ \phi \ A' \Leftrightarrow \exists \alpha. \, \textsc{Nuprl}_j A A' \alpha \\
\textsc{Nuprl}_i & \stackrel{\text{def}}{=} & \textsc{Close}(\textsc{Univ}_i)
\end{array}
$$

Finally we may define a candidate type system $\textsc{Univ}_\omega$ that defines all universes, and define the desired $\textsc{Nuprl}_\omega$ to be its closure:

$$
\begin{array}{rll}
\textsc{Univ}_\omega TT'\phi & \textit{iff} & \exists i. \, T \Downarrow \mathbb{U}_i \wedge T' \Downarrow \mathbb{U}_i \wedge \\
& & \qquad \forall A, A'. \, A \ \phi \ A' \Leftrightarrow \exists \alpha. \, \textsc{Nuprl}_i A A' \alpha \\
\textsc{Nuprl}_\omega & \stackrel{\text{def}}{=} & \textsc{Close}(\textsc{Univ}_\omega)
\end{array}
$$

We may now define the desired equality relations using $\textsc{Nuprl}_\omega$:

Definition 4.9

- $T_1 = T_2$ *iff* $\exists \phi. \, \textsc{Nuprl}_\omega T_1 T_2 \phi$
- $t_1 = t_2 \in T$ *iff* $\exists \phi. \, \textsc{Nuprl}_\omega T T \phi \wedge t_1 \ \phi \ t_2$

It remains to show that the equality relations have the desired properties. First we must show that $\textsc{Nuprl}_\omega$ is actually a type system, as intended. Then we will observe that $\textsc{Nuprl}_\omega$ adheres to the specification in Figure 4.5.

Lemma 4.10 *If τ is a type system, then for each* $\textsc{op} \in \textsc{Types}$, $\textsc{op}(\tau)$ *is a type system.*

Lemma 4.11 *Each operator in* $\textsc{Types}$ *defines disjoint types from the others and from τ (when τ defines only universes). Formally, if*

$$
\begin{array}{lll}
\textsc{Set}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', \alpha, \beta_{(-)}, \beta'_{(-)}, e, e'. \\
 & & \quad T \Downarrow \{x : A \mid B\} \wedge T \Downarrow \{x : A' \mid B'\} \wedge \tau AA'\alpha \wedge \\
 & & \quad \forall a, a'.\, a \;\alpha\; a' \Rightarrow \tau(B[a/x])(B[a'/x])\beta_a \wedge \\
 & & \quad \forall a, a'.\, a \;\alpha\; a' \Rightarrow \tau(B'[a/x])(B'[a'/x])\beta'_a \wedge \\
 & & \quad (B \text{ implies } B') \\
 & & \quad \forall a, b, a', b'.\, a \;\alpha\; a' \Rightarrow b \;\beta_a\; b' \Rightarrow e[a, b/x, y] \;\beta'_a\; e[a', b'/x, y] \wedge \\
 & & \quad (B' \text{ implies } B) \\
 & & \quad \forall a, b, a', b'.\, a \;\alpha\; a' \Rightarrow b \;\beta'_a\; b' \Rightarrow e'[a, b/x, y] \;\beta_a\; e'[a', b'/x, y] \wedge \\
 & & \quad \forall t, t'.\, t \;\phi\; t' \Leftrightarrow (t \;\alpha\; t' \wedge \exists b.\, b \;\beta_t\; b) \\
\textsc{Quot}(\tau)TT'\phi & \textit{iff} & \exists A, B, A', B', \alpha, \beta_{(-,-)}, \beta'_{(-,-)}, e, e', e_r, e_s, e_t. \\
 & & \quad T \Downarrow xy{:}A//B \wedge T \Downarrow xy{:}A'//B' \wedge \tau AA'\alpha \wedge \\
 & & \quad \forall a_1, a_2, a'_1, a'_2.\, a_1 \;\alpha\; a'_1 \Rightarrow a_2 \;\alpha\; a'_2 \Rightarrow \\
 & & \qquad \tau(B[a_1, a_2/x, y])(B[a'_1, a'_2/x, y])\beta_{(a_1,a_2)} \wedge \\
 & & \quad \forall a_1, a_2, a'_1, a'_2.\, a_1 \;\alpha\; a'_1 \Rightarrow a_2 \;\alpha\; a'_2 \Rightarrow \\
 & & \qquad \tau(B'[a_1, a_2/x, y])(B'[a'_1, a'_2/x, y])\beta'_{(a_1,a_2)} \wedge \\
 & & \quad (B \text{ implies } B') \\
 & & \quad \forall a_1, a_2, a'_1, a'_2, b, b'.\, a_1 \;\alpha\; a'_1 \Rightarrow a_2 \;\alpha\; a'_2 \Rightarrow b \;\beta_{(a_1,a_2)}\; b' \Rightarrow \\
 & & \qquad e[a_1, a_2, b/x, y, z] \;\beta'_{(a_1,a_2)}\; e[a'_1, a'_2, b'/x, y, z] \wedge \\
 & & \quad (B' \text{ implies } B) \\
 & & \quad \forall a_1, a_2, a'_1, a'_2, b, b'.\, a_1 \;\alpha\; a'_1 \Rightarrow a_2 \;\alpha\; a'_2 \Rightarrow b \;\beta'_{(a_1,a_2)}\; b' \Rightarrow \\
 & & \qquad e[a_1, a_2, b/x, y, z] \;\beta_{(a_1,a_2)}\; e[a'_1, a'_2, b'/x, y, z] \wedge \\
 & & \quad (\text{reflexivity}) \\
 & & \quad \forall a, a'.\, a \;\alpha\; a' \Rightarrow e_r[a/x] \;\beta_{(a,a)}\; e_r[a'/x] \wedge \\
 & & \quad (\text{symmetry}) \\
 & & \quad \forall a_1, a_2, a'_1, a'_2, b, b'.\, a_1 \;\alpha\; a'_1 \Rightarrow a_2 \;\alpha\; a'_2 \Rightarrow b \;\beta_{(a_1,a_2)}\; b' \Rightarrow \\
 & & \qquad e_s[a_1, a_2, b/x, y, z] \;\beta_{(a_2,a_1)}\; e_s[a'_1, a'_2, b'/x, y, z] \wedge \\
 & & \quad (\text{transitivity}) \\
 & & \quad \forall a_1, a_2, a_3, a'_1, a'_2, a'_3, b_1, b_2, b'_1, b'_2. \\
 & & \qquad a_1 \;\alpha\; a'_1 \Rightarrow a_2 \;\alpha\; a'_2 \Rightarrow a_3 \;\alpha\; a'_3 \Rightarrow \\
 & & \qquad b_1 \;\beta_{(a_1,a_2)}\; b'_1 \Rightarrow b_2 \;\beta_{(a_2,a_3)}\; b'_2 \Rightarrow \\
 & & \qquad e_t[a_1, a_2, a_3, b_1, b_2/x, y, z, v, w] \;\beta_{(a_1,a_3)} \\
 & & \qquad\qquad e_s[a'_1, a'_2, a'_3, b'_1, b'_2/x, y, z, v, w] \wedge \\
 & & \quad \forall t, t'.\, t \;\phi\; t' \Leftrightarrow (t \;\alpha\; t \wedge t' \;\alpha\; t' \wedge \exists b.\, b \;\beta_{(t,t')}\; b)
\end{array}
$$

Figure 4.8: Type Definitions (Set and Quotient)

- *τ is type symmetric and type transitive, and*
- *τ defines only universes (that is, if $\tau TT\phi$ then $T \Downarrow \mathbb{U}_i$ for some i), and*
- $\textsc{Close}(\tau)T_1T'_1\phi_1$ *and* $\textsc{Close}(\tau)T_2T'_2\phi_2$*, and*
- *either $T_1 \equiv T_2$ or $T_1 \equiv T'_2$,*

then either $\tau T_1T'_1\phi_1$ and $\tau T_2T'_2\phi_2$, or for some $\textsc{op} \in \textsc{Types}$, $\textsc{op}(\textsc{Close}(\tau))T_1T'_1\phi_1$ *and* $\textsc{op}(\textsc{Close}(\tau))T_2T'_2\phi_2$.

Lemma 4.12 *If τ is a type system and if τ defines only universes, then* $\textsc{Close}(\tau)$ *is a type system.*

Lemma 4.13 *For all i,* $\textsc{Univ}_i$ *and* $\textsc{Nuprl}_i$ *are type systems.*

Lemma 4.14 Univ_ω *and* Nuprl_ω *are type systems.*

Proof

First show that Univ_ω is a type system: All but unique valuation and type transitivity are directly by Lemma 4.13. For unique valuation and type transitivity, use Lemma 4.13 with the observation that Univ_i is cumulative: for all $i < j$, if $\text{Univ}_i TT'\phi$ then $\text{Univ}_j TT'\phi$. Then Nuprl_ω is a type system by Lemma 4.12. □

Proposition 4.15 *The equality relations $T_1 = T_2$ and $t_1 = t_2 \in T$ adhere to the specification in Figure 4.5.*

4.4.2 Judgement Semantics

I begin the semantics for judgements by defining the notion of an assignment of closed terms to variables:

Definition 4.16

- *An* assignment *is a sequence of pairs $\langle x_1, t_1\rangle \cdots \langle x_n, t_n\rangle$ such that each x_i is a distinct variable and each t_i is a closed term.*
- *The application $\alpha(t)$ of an assignment α to a term t, where $\alpha \equiv \langle x_1, t_1\rangle \cdots \langle x_n, t_n\rangle$, is $t[t_1 \cdots t_n / x_1 \cdots x_n]$.*

As discussed in Section 4.2.1, the intended meaning of the sequent $H \vdash_\nu C \triangleleft t$ is that $\alpha(C) = \alpha'(C)$ and $\alpha(t) = \alpha'(t) \in \alpha(C)$ when α and α' are *equal assignments* for the hypotheses H. In order to formalize this as a semantics for judgements, I must define exactly what is meant by equal assignments.

The obvious idea is to say that assignments are equal when their corresponding terms are equal in the appropriate types. We may call this notion "assignment similarity":

Definition 4.17 *Assignment similarity (written $\alpha_1 \approx \alpha_2 \in H$) is the smallest relation such that:*

- $\epsilon \approx \epsilon \in \epsilon$
- $\alpha_1\langle x, t_1\rangle \approx \alpha_2\langle x, t_2\rangle \in (H; x : T)$ *if* $\alpha_1 \approx \alpha_2 \in H$ *and* $t_1 = t_2 \in \alpha_1(T)$

This notion provides a good starting point, but it will not serve as the needed notion of assignment equality, because it is neither symmetric nor transitive. Symmetry does not hold because it is not generally the case for similar assignments α_1 and α_2 that $\alpha_1(T) = \alpha_2(T)$, so $t_1 = t_2 \in \alpha_1(T)$ does not imply $t_2 = t_1 \in \alpha_2(T)$. Transitivity fails for the same reason.

The solution is to impose on the inductive case the additional requirement that $\alpha_1(T) = \alpha_2(T)$. The leads to the first of three definitions of assignment equality:

Definition 4.18 *Assignment equality with minimal functionality (written $\alpha_1 = \alpha_2 \in_{min} H$) is the smallest relation such that:*

- $\epsilon = \epsilon \in_{min} \epsilon$
- $\alpha_1\langle x, t_1\rangle = \alpha_2\langle x, t_2\rangle \in_{min} (H; x : T)$ *if*

- $\alpha_1 = \alpha_2 \in_{min} H$, *and*
- $t_1 = t_2 \in \alpha_1(T)$, *and*
- $\alpha_1(T) = \alpha_2(T)$

Proposition 4.19 *Assignment equality with minimal functionality is symmetric and transitive.*

This notion of equality is called *minimal functionality* because it uses the minimum extra constraint (over similarity) to ensure symmetry and transitivity. By strengthening that constraint, we can get two other interesting forms of assignment equality. Note that in the third clause of the inductive case of Definition 4.18, the assignments α_1 and α_2 are similar (since clearly $\alpha_1 = \alpha_2 \in_{min} H$ implies $\alpha_1 \approx \alpha_2 \in H$) and fixed. We define *pointwise functionality* by allowing one of the similar assignments to vary, and we define *full functionality* by allowing both of the similar assignments to vary:

Definition 4.20 *Assignment equality with pointwise functionality (written $\alpha_1 = \alpha_2 \in_{pw} H$) is the smallest relation such that:*

- $\epsilon = \epsilon \in_{pw} \epsilon$
- $\alpha_1\langle x, t_1\rangle = \alpha_2\langle x, t_2\rangle \in_{pw} (H; x : T)$ *if*
  - $\alpha_1 = \alpha_2 \in_{pw} H$, *and*
  - $t_1 = t_2 \in \alpha_1(T)$, *and*
  - *for all* α, *if* $\alpha_1 \approx \alpha \in H$ *then* $\alpha_1(T) = \alpha(T)$

Definition 4.21 *Assignment equality with full functionality (written $\alpha_1 = \alpha_2 \in_{full} H$) is the smallest relation such that:*

- $\epsilon = \epsilon \in_{full} \epsilon$
- $\alpha_1\langle x, t_1\rangle = \alpha_2\langle x, t_2\rangle \in_{full} (H; x : T)$ *if*
  - $\alpha_1 = \alpha_2 \in_{full} H$, *and*
  - $t_1 = t_2 \in \alpha_1(T)$, *and*
  - *for all* α, α', *if* $\alpha \approx \alpha' \in H$ *then* $\alpha(T) = \alpha'(T)$

Following are some important properties of pointwise and full functionality. For any of the three forms of functionality, I write $\alpha \in H$ to mean $\alpha = \alpha \in H$.

Proposition 4.22

- *If* $\alpha \in_{pw} H$, *then for any* α', $\alpha \approx \alpha' \in H$ *implies* $\alpha = \alpha' \in_{pw} H$
- *If* $\alpha \in_{full} H$, *then for any* α_1, α_2, $\alpha_1 \approx \alpha_2 \in H$ *implies* $\alpha_1 = \alpha_2 \in_{full} H$
- *Assignment equality with pointwise functionality is symmetric and transitive.*
- *Assignment equality with full functionality is symmetric and transitive.*

We can now define the semantics of Nuprl judgements. Following Constable *et al.* [19], I use pointwise functionality. (An avenue for future research is to explore semantics based on full [6] or minimal functionality.) The semantics given by Constable *et al.* [19] is equivalent to this one, but is considerably more complicated.

Definition 4.23 *The judgement $H \vdash_\nu C \lhd t$ is valid if for all assignments α and α', $\alpha = \alpha' \in_{pw} H$ implies $\alpha(C) = \alpha'(C)$ and $\alpha(t) = \alpha'(t) \in \alpha(C)$. The judgement $\vdash_\nu t \sim t'$ is valid if $t \sim t'$.*

Theorem 4.24 (Nuprl Soundness) *Any judgement derivable in the Nuprl proof rules is valid.*

Proof

By induction on derivations.

Corollary 4.25 (Nuprl Consistency) *The Nuprl proof rules are consistent.*

Proof

The judgement $\vdash_\nu$ *Void* $\lhd \star$ is invalid, and therefore unprovable. □

4.4.3 Classical Reasoning

As discussed in Section 4.1.5, classical reasoning principles are semantically invalid within the type theory. In the Nuprl semantics, validity of the excluded middle principle ($\forall P. P \vee \neg P$) would imply the existence of a computable function that would determine the truth or falsity of every proposition. However, in the underlying metatheory, we may consider using classical reasoning principles.

The proof rules given in Appendix B and not marked as classical may all be proven valid without classical reasoning. However, there are some additional rules that may be shown valid in a classical metatheory.

The primary such rule is a weak form of the excluded middle principle:

$$\frac{}{H \vdash \forall P{:}\mathbb{P}_i.\, \downarrow\!(P \vee \neg P) \lhd \lambda x.\star} \text{(XM)}$$

Proposition 4.26 *The rule XM is semantically valid in a classical interpretation.*

Note that this works because of the squashing of $P \vee \neg P$. The alternative axiom $\downarrow\!(\forall P{:}\mathbb{P}_i.\, P \vee \neg P)$ is not semantically valid, because it still implies the existence of a general decision function, even though that function is still not provided. In Chapter 5 we will see another example of important rules validated by a classical metatheory but not a constructive metatheory. These are the coadmissibility rules for partial types (Rules PWC and PC).

One practical use of the XM rule is for fixpoint induction (Section 4.3.3). Recall that the fixpoint induction rule generates the subgoal $H; x : \overline{T}; y : (x\ in!\ T \Rightarrow P) \vdash_\nu P$. This subgoal may be proven from the simpler subgoal $H \vdash_\nu P[\bot/x]$ using the XM rule (and some of the other subgoals of the fixpoint induction rule): Instantiating XM gives $\downarrow\!(x\ in!\ T \vee \neg(x\ in!\ T))$. Then $\downarrow\! P$ can be proven by cases, invoking hypothesis y if $x\ in!\ T$ and substituting $\bot$ for x if $\neg(x\ in!\ T)$. By subgoal 8, we may conclude P from $\downarrow\! P$.

Chapter 5

Admissibility

One of the earliest logical theorem provers was the LCF system [38], based on the logic of partial computable functions [88, 89]. Although LCF enjoyed many groundbreaking successes, one problem it faced was that, although it supported a natural notion of *partial* function, it had difficulty expressing the notion of a *total* function. Later theorem provers based on constructive type theory, such as Nuprl [19], based on Martin-Löf type theory [68], and Coq [8], based on the Calculus of Constructions [29], faced the opposite problem; they had a natural notion of total functions, but had difficulty dealing with partial functions. The lack of partial functions seriously limited the scope of those theorem provers, because it made them unable to reason about programs in real programming languages where recursion does not always necessarily terminate.

This problem may be addressed in type theory by the partial type discussed in Chapter 4. In a partial type theory, recursively defined objects may be typed using the *fixpoint principle:* if f has type $\overline{T} \to \overline{T}$ then $\mathit{fix}(f)$ has type $\overline{T}$. However, the fixpoint principle is not valid for every type T; it is only valid for types that are *admissible.* This phenomenon was not unknown to LCF; LCF used the related device of fixpoint induction, which was valid only for admissible predicates. When the user attempted to invoke fixpoint induction, the system would automatically check that the goal was admissible using a set of syntactic rules [55].

Despite their obvious uses in program analysis, partial types have seen little use in theorem proving systems [25, 9, 7]. This is due in large part to two problems: partial type extensions have been developed only for fragments of type theory that do not include equality, and too few types have been known to be admissible. I addressed the former problem in Chapter 4; in this chapter I address the latter.

Smith [92] gave a significant class of admissible types for a Nuprl-like theory, but his class required product types to be non-dependent. The type $\Sigma x{:}A.B$ (where x appears free in B) was explicitly excluded. Later, Smith [91] extended his class to include some dependent products $\Sigma x{:}A.B$, but disallowed any free occurrences of x to the left of an arrow in B. Partial type extensions to Coq [7] were also restrictive, assuming function spaces to be the only type constructor. These restrictions are quite strong; dependent products are used in encodings of modules [65], objects [83], algebras [57], and even such simple devices as variant records. Furthermore, ruling out dependent products disallows reasoning using fixpoint induction as in LCF. (This is explained further in Section 5.1.) Finally, the restriction is particularly unsatisfying since most types used in practice do turn out to be admissible, and may be shown so by metatheoretical reasoning.

In this chapter I present a very wide class of admissible types using two devices, a condition called *predicate-admissibility* and a monotonicity condition. In particular, many dependent products may be shown to be admissible. Predicate-admissibility relates to when the limit of a chain of type approximations contains certain terms, whereas admissibility relates to the membership of a single type. Monotonicity is a simpler condition that will be useful for showing types admissible that do not involve partiality.

5.1 The Fixpoint Principle

The central issue of this chapter is the *fixpoint principle:*

$$f \in \overline{T} \to \overline{T} \Rightarrow \mathit{fix}(f) \in \overline{T}$$

The fixpoint principle allows us to type recursively defined objects, such as recursive functions. Unfortunately, unlike in programming languages, where the principle can usually be invoked on arbitrary types, expressive type theories such as the one in this thesis contain types for which the fixpoint principle is not valid. I shall informally say that a type is *admissible* if the fixpoint principle is valid for that type and give a formal definition in Section 5.3. To make maximum use of a partial type theory, one wants as large a class of admissible types as possible.

In Section 5.3 I will explore two wide classes of admissible types, one derived from a *predicate-admissibility* condition and another derived from a monotonicity condition. But first, it is worthwhile to note that there are indeed inadmissible types:

Theorem 5.1 *There exist inadmissible types.*

Proof Sketch

This example is due to Smith [92]. Recall that $\mathbb{N} = \{n : \mathbb{Z} \mid 0 \leq n\}$. Let T be the type of functions that do not halt for all inputs, and let f be the function that halts on zero, and on any other n immediately recurses with $n - 1$. This is formalized as follows:

$$\begin{array}{rcl} T & \stackrel{\text{def}}{=} & \Sigma h{:}(\mathbb{N} \to \overline{\mathbb{N}}).\,((\Pi x{:}\mathbb{N}.\, h\ x\ \mathit{in}!\ \mathbb{N}) \to \mathit{Void}) \\ f & \stackrel{\text{def}}{=} & \lambda p.\langle \lambda x.\, \mathit{if}\ x \leq_Z 0\ \mathit{then}\ 0\ \mathit{else}\ \pi_1(p)(x-1), \lambda y.\star\rangle \end{array}$$

Intuitively, any finite approximation of $\mathit{fix}(f)$ will recurse some limited number of times and then give up, placing it in T, but $\mathit{fix}(f)$ will halt for every input, excluding it from T. Formally, the function f has type $\overline{T} \to \overline{T}$, but $\mathit{fix}(f) \notin \overline{T}$. (The proof of these two facts appears in Appendix C.) Therefore T is not admissible.

5.2 Computational Lemmas

Before presenting my main results in Section 5.3, I first require some lemmas about the computational behavior of the fixpoint operator. The central result is that $\mathit{fix}(f)$ is the least upper bound of the finite approximations $\bot, f(\bot), f(f(\bot)), \ldots$ with regard to a computational approximation relation defined below. The compactness of *fix* (if $\mathit{fix}(f)$ halts then one of its finite approximations halts) will be a simple corollary of this result. However, the proof of the least upper bound theorem is considerably more elegant than most proofs of compactness.

5.2.1 Computational Approximation

For convenience, throughout this section we will frequently consider terms using a unified representation scheme for terms: A term is either a variable or a compound term $\theta(x_{11}\cdots x_{1k_1}.t_1,\ldots, x_{n1}\cdots x_{nk_n}.t_n)$ where the variables $x_{i1},\ldots,x_{ik_i}$ are bound in the subterm t_i. For example, the term $\Pi x{:}T_1.T_2$ is represented $\Pi(T_1, x.T_2)$ and the term $\langle t_1, t_2\rangle$ is represented $\langle\rangle(t_1, t_2)$.

Informally speaking, a term t_1 approximates the term t_2 when: if t_1 converges to a canonical form then t_2 converges to a canonical form with the same outermost operator, and the subterms of t_1's canonical form approximate the corresponding subterms of t_2's canonical form. The formal definition appears below and is due to Howe [53].[1] Following Howe, when R is a binary relation on closed terms, I adopt the convention extending R to possibly open terms that if t and t' are possibly open then $t\ R\ t'$ if and only if $\sigma(t)\ R\ \sigma(t')$ for every substitution σ such that $\sigma(t)$ and $\sigma(t')$ are closed.

Definition 5.2 (Computational Approximation)

- *Let R be a binary relation on closed terms and suppose e and e' are closed. Then $e\ C(R)\ e'$ exactly when if $e \Downarrow \theta(\vec{x}_1.t_1,\ldots,\vec{x}_n.t_n)$ then there exists some closed $e'' = \theta(\vec{x}_1.t'_1,\ldots,\vec{x}_n.t'_n)$ such that $e' \Downarrow e''$ and $t_i\ R\ t'_i$.*
- *$e \leq_0 e'$ whenever e and e' are closed.*
- *$e \leq_{i+1} e'$ if and only if $e\ C(\leq_i)\ e'$*
- *$e \leq e'$ if and only if $e \leq_i e'$ for every i*

Definition 5.3 (Computational Equivalence) *The terms e and e' are computationally equivalent ($e \sim e'$) if and only if $e \leq e'$ and $e' \leq e$.*

The following are facts about computational approximation that will be used without explicit reference. The first two follow immediately from the definition, the third is easy using determinism (Proposition 4.2) and the last is proven using Howe's method [53].

Proposition 5.4

- *$\leq$ and $\leq_i$ are reflexive and transitive.*
- *If $t \mapsto t'$ then $t' \leq t$ and $t' \leq_i t$.*
- *If $t \mapsto t'$ then $t \leq t'$ and $t \leq_i t'$.*

Lemma 5.5 (Congruence) *If $e \leq e'$ and $t \leq t'$ then $e[t/x] \leq e'[t'/x]$.*

[1] Howe's definition actually differs slightly from the one here; he defines $\leq$ as the greatest fixed point of the operator C. It is not difficult to show that the two definitions are equivalent, as long as the computation system is deterministic (Proposition 4.2). If the computation system is nondeterministic, the definition here fails to be a fixed point, and the more complicated greatest fixed point definition must be employed.

5.2.2 Finite Approximations

With this notion of computational approximation in hand, we may now show that the terms $\perp, f\,\perp, f(f\,\perp), \ldots$ form a chain of approximations to the term $\mathit{fix}(f)$. Let $\perp$ be the divergent term $\mathit{fix}(\lambda x.x)$. Since $\perp$ never converges, $\perp \leq t$ for any term t. Let f^i be defined as follows:

$$\begin{aligned} f^0 &\stackrel{\text{def}}{=} \perp \\ f^{i+1} &\stackrel{\text{def}}{=} f(f^i) \end{aligned}$$

Certainly $f^0 \leq f^1$, since $f^0 \equiv \perp$. By congruence, $f(f^0) \leq f(f^1)$, and thus $f^1 \leq f^2$. Similarly, $f^i \leq f^{i+1}$ for all i. Thus $f^0, f^1, f^2, \ldots$ forms a chain; I now wish to show that $\mathit{fix}(f)$ is an upper bound of the chain. Certainly $f^0 \leq \mathit{fix}(f)$. Suppose $f^i \leq \mathit{fix}(f)$. By congruence $f(f^i) \leq f(\mathit{fix}(f))$. Thus, since $\mathit{fix}(f) \mapsto f(\mathit{fix}(f))$, it follows that $f^{i+1} \equiv f(f^i) \leq f(\mathit{fix}(f)) \leq \mathit{fix}(f)$. By induction it follows that $\mathit{fix}(f)$ is an upper bound of the chain. The following corollary follows from congruence and the definition of approximation:

Corollary 5.6 *If there exists j such that $e[f^j/x]\downarrow$ then $e[\mathit{fix}(f)/x]\downarrow$. Moreover, the canonical forms of $e[f^j/x]$ and $e[\mathit{fix}(f)/x]$ must have the same outermost operator.*

5.2.3 Least Upper Bound Theorem

In this section I summarize the proof of the least upper bound theorem. To begin, we need a lemma stating a general property of evaluation. Lemma 5.7 captures the intuition that closed, noncanonical terms that lie within a term being evaluated are not destructed; they either are moved around unchanged (the lemma's first case) or are evaluated in place with the surrounding term left unchanged (the lemma's second case). The variable x indicates positions where the term of interest is found and, in the second case, the variable y indicates which of those positions, if any, is about to be evaluated.

Lemma 5.7 *If $e_1[t/x] \mapsto e_2$, and $e_1[t/x]$ is closed, and t is closed and noncanonical, then either*

- *there exists e_2' such that for any closed t', $e_1[t'/x] \mapsto e_2'[t'/x]$, or*
- *there exist e_1' and t' such that $e_1 \equiv e_1'[x/y]$, $t \mapsto t'$ and for any closed t'', $e_1'[t'', t/x, y] \mapsto e_1'[t'', t'/x, y]$.*

It is worthwhile to note that Proposition 4.2 and Lemmas 5.5 and 5.7 are the only properties of evaluation used in the proof of the least upper bound theorem, and that these properties are true in computational systems with considerable generality. Consequently, the theorem may be used in a variety of applications beyond the computational system of this thesis.

Lemma 5.8 shows that *fix* terms may be effectively simulated in any particular computation by sufficiently large finite approximations. The lemma is simplified by using computational approximation instead of evaluation for the simulation, which makes it unnecessary to track which of the approximations are unfolded and which are not, an issue that often complicates compactness proofs.

Lemma 5.8 (Simulation) *For all f, e_1 and e_2 (where f is closed and x is the only free variable of e_1), there exist j and e_2' such that if $e_1[\mathit{fix}(f)/x] \mapsto^* e_2$ then $e_2 \equiv e_2'[\mathit{fix}(f)/x]$ and for all $k \geq j$, $e_2'[f^{k-j}/x] \leq e_1[f^k/x]$.*

Theorem 5.9 (Least Upper Bound) *For all f, t and e (where f is closed), if $\forall j.\, e[f^j/x] \leq t$, then $e[\mathit{fix}(f)/x] \leq t$.*

Proof Sketch

By induction on l that $e[\mathit{fix}(f)/x] \leq_l t$. (The complete proof in Appendix C addresses free variables.) Suppose $e[\mathit{fix}(f)/x]$ evaluates to some canonical form $e'[\mathit{fix}(f)/x]$ (where e' is chosen by Lemma 5.8). Let e' be of the form $\theta(\vec{x}_1.t_1, \ldots, \vec{x}_n.t_n)$. Using Lemma 5.8, the assumption $\forall k.\, e[f^k/x] \leq t$, and transitivity, we may show that $e'[f^j/x] \leq t$ for all j. Therefore $t \Downarrow \theta(\vec{x}_1.t'_1, \ldots, \vec{x}_n.t'_n)$ and $t_i[f^j/x] \leq t'_i$ for all j. Now, by induction, $t_i[\mathit{fix}(f)/x] \leq_l t'_i$. Thus $e[\mathit{fix}(f)/x] \leq_{l+1} t$.

There are two easy corollaries to the least upper bound theorem. One is that $\mathit{fix}(f)$ is the least fixed point of f, and the other is compactness.

Corollary 5.10 (Least Fixed Point) *For all closed f and t, if $f(t) \leq t$ then $\mathit{fix}(f) \leq t$.*

Proof

Certainly $f^0 \equiv \bot \leq t$. Then $f^1 \equiv f(f^0) \leq f(t) \leq t$. Similarly, by induction, $f^j \leq t$ for any j. Therefore $\mathit{fix}(f) \leq t$ by Theorem 5.9. □

Corollary 5.11 (Compactness) *If f is closed and $e[\mathit{fix}(f)/x]\downarrow$ then there exists some j such that $e[f^j/x]\downarrow$. Moreover, the canonical forms of $e[\mathit{fix}(f)/x]$ and $e[f^j/x]$ must have the same outermost operator.*

Proof

Suppose there does not exist j such that $e[f^j/x]\downarrow$. Then $e[f^j/x] \leq \bot$ for all j. By Theorem 5.9, $e[\mathit{fix}(f)/x] \leq \bot$. Therefore $e[\mathit{fix}(f)/x]$ does not converge, but this contradicts the assumption,[2] so there must exist j such that $e[f^j/x]\downarrow$. Since $e[f^j/x] \leq e[\mathit{fix}(f)/x]$, the canonical forms of $e[f^j/x]$ and $e[\mathit{fix}(f)/x]$ must have the same outermost operator. □

5.3 Admissibility

I am now ready to begin specifying some wide classes of types for which the fixpoint principle is valid. First we define admissibility. The simple property of validating the fixpoint principle is too specific to allow any good closure conditions to be shown easily, so we generalize a bit to define admissibility. A type is *admissible* if the upper bound $t[\mathit{fix}(f)]$ of an approximation chain $t[f^0], t[f^1], t[f^2], \ldots$ belongs to the type whenever a cofinite subset of the chain belongs to the type. This is formalized as Definition 5.13, but first I define some convenient notation.

Notation 5.12 *For any natural number j, the notation $t^{[j]_f}$ means $t[f^j/w]$, and the notation $t^{[\omega]_f}$ means $t[\mathit{fix}(f)/w]$. Also, the f subscript is dropped when the intended term f is unambiguously clear.*

Definition 5.13 *A type T is* admissible *(abbreviated* Adm(T)*) if:*

$$\forall f, t, t'.\, (\exists j.\, \forall k \geq j.\, t^{[k]} = t'^{[k]} \in T) \Rightarrow t^{[\omega]} = t'^{[\omega]} \in T$$

[2]Although this proof is non-constructive, a slightly less elegant constructive proof may be derived directly from Lemma 5.8.

As expected, admissibility is sufficient to guarantee applicability of the fixpoint principle:

Theorem 5.14 *For any T and f, if T is admissible and $f = f' \in \overline{T} \rightarrow \overline{T}$ then $\mathit{fix}(f) = \mathit{fix}(f') \in \overline{T}$.*

Proof

$\overline{T}$ type since $\overline{T} \rightarrow \overline{T}$ type. Note that $f^j = f'^j \in \overline{T}$ for every j. Suppose $\mathit{fix}(f)\downarrow$. By compactness, $f^j\downarrow$ for some j. Since $f^j = f'^j \in \overline{T}$, it follows that $f'^j\downarrow$ and thus $\mathit{fix}(f')\downarrow$ by Corollary 5.6. Similarly $\mathit{fix}(f')\downarrow$ implies $\mathit{fix}(f)\downarrow$. It remains to show that $\mathit{fix}(f) = \mathit{fix}(f') \in T$ when $\mathit{fix}(f)\downarrow$. Suppose again that $\mathit{fix}(f)\downarrow$. As before, there exists j such that $f^j\downarrow$ by compactness. Hence $f^j = f'^j \in T$. Since T is admissible, $\mathit{fix}(f) = \mathit{fix}(f') \in T$. $\square$

A number of closure conditions exist on admissible types and are given in Lemma 5.15. Informally, basic compound types other than dependent products are admissible so long as their component types in positive positions are admissible. Base types—natural numbers, convergence types, and (for this lemma only) equality types—are always admissible. These are essentially the admissible types of Smith [92], except that for a function type to be admissible Smith required that its domain type be admissible.

Lemma 5.15

- $\mathrm{Adm}(A + B)$ *if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B)$
- $\mathrm{Adm}(\Pi x{:}A.B)$ *if* $\forall a \in A.\, \mathrm{Adm}(B[a/x])$
- $\mathrm{Adm}(A \times B)$ *if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B)$
- $\mathrm{Adm}(\mathit{Void})$, $\mathrm{Adm}(\mathit{Atom})$, $\mathrm{Adm}(\mathbb{Z})$ *and* $\mathrm{Adm}(\mathbb{E})$
- $\mathrm{Adm}(a = a' \mathit{\ in\ } A)$
- $\mathrm{Adm}(a \leq a')$
- $\mathrm{Adm}(\overline{A})$ *if* $\mathrm{Adm}(A)$
- $\mathrm{Adm}(a \mathit{\ in!\ } A)$

Proof

The proof follows the same lines as Smith's proof, except that handling equality adds a small amount of complication to the proof. I show the function case by way of example.

Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \Pi x{:}A.B$. I need to show that $t^{[\omega]} = t'^{[\omega]} \in \Pi x{:}A.B$. Since $\Pi x{:}A.B$ is inhabited it is a type. Both $t^{[j]}$ and $t'^{[j]}$ converge to lambda abstractions, so, by Corollary 5.6, $t^{[\omega]} \Downarrow \lambda x.b$ and $t'^{[\omega]} \Downarrow \lambda x.b'$ for some terms b and b'. Suppose $a = a' \in A$. To get that $b[a/x] = b'[a'/x] \in B[a/x]$ it suffices to show that $t^{[\omega]}a = t'^{[\omega]}a' \in B[a/x]$. Since $\mathrm{Adm}(B[a/x])$, it suffices to show that $\forall k \geq j.\, t^{[k]}a = t'^{[k]}a' \in B[a/x]$, which follows from the supposition. $\square$

Unfortunately, Lemma 5.15 can show the admissibility of a product space only if it is *non-dependent.* Dependent products do not have an admissibility condition similar to that of dependent functions. This reason for this is as follows: Admissibility states that a *single fixed type* contains the limit of an approximation chain if it contains a cofinite subset of that chain. For functions, disjoint union, partial types, and non-dependent products it is possible to decompose prospective members in such a way that admissibility may be applied to a single type (such as the type $B[a/x]$ used in the proof of Lemma 5.15). In contrast, for a dependent product, the right-hand term's desired type depends upon the left-hand term, which is changing at the same time as the right-hand term. Consequently, there is no single type into which to place the right-hand term.

However, understanding the problem with dependent products suggests a solution, to generalize the definition of admissibility to allow the type to vary. This leads to the notion of *predicate-admissibility* that I discuss in the next section.

5.3.1 Predicate-Admissibility

Definition 5.16 *A type T is* predicate-admissible *for x in S (abbreviated* $\mathrm{Adm}(T \mid x : S)$*) if:*

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j. \forall k \geq j.\, e^{[k]} \in S \wedge t^{[k]} = t'^{[k]} \in T[e^{[k]}/x]) \Rightarrow t^{[\omega]} = t'^{[\omega]} \in T[e^{[\omega]}/x]$$

The term "predicate-admissibility" stems from its similarity to the notion of admissibility of predicates in domain theory (and LCF). If one ignores the inhabiting terms t and t', which may be seen as evidences of the truth of the predicate $T[\,]$, then predicate-admissibility is saying $T[e^{[\omega]}]$ if $T[e^{[k]}]$ for all k greater than some j. This is precisely the notion of admissibility of predicates in domain theory. Indeed, the results here were influenced by the work of Igarashi [55], who established conditions on admissibility of domain-theoretic predicates.

To show the admissibility of a dependent product type, it is sufficient to show predicate-admissibility of the right-hand side (along with admissibility of the left):

Lemma 5.17 *The type $\Sigma x{:}A.B$ is admissible if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B \mid x : A)$.

Proof

Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \Sigma x{:}A.B$. It is necessary to show that $t^{[\omega]} = t'^{[\omega]} \in \Sigma x{:}A.B$. Since $\Sigma x{:}A.B$ is inhabited it is a type. Both $t^{[j]}$ and $t'^{[j]}$ converge to pairs, so, by Corollary 5.6, $t^{[\omega]} \Downarrow \langle a, b\rangle$ and $t'^{[\omega]} \Downarrow \langle a', b'\rangle$ for some terms a, b, a' and b'. To get that $a = a' \in A$ it suffices to show that $\pi_1(t^{[\omega]}) = \pi_1(t'^{[\omega]}) \in A$. Since $\mathrm{Adm}(A)$, it suffices to show that $\forall k \geq j.\, \pi_1(t^{[k]}) = \pi_1(t'^{[k]}) \in A$, which follows from the supposition.

To get that $b = b' \in B[a/x]$ (the interesting part), it suffices to show that $\pi_2(t^{[\omega]}) = \pi_2(t'^{[\omega]}) \in B[\pi_1(t^{[\omega]})/x]$. Since $\mathrm{Adm}(B \mid x : A)$, it suffices to show that $\pi_1(t^{[\omega]}) \in A$, which has already been shown, and $\forall k \geq j.\, \pi_1(t^{[k]}) \in A \wedge \pi_2(t^{[k]}) = \pi_2(t'^{[k]}) \in B[\pi_1(t^{[k]})/x]$, which follows from the supposition. □

The conditions for predicate-admissibility are more elaborate, but also more general. I may immediately state conditions for basic types other than functions. Informally, basic compound types other than functions are predicate-admissible so long as their component types are predicate-admissible, and base types are always predicate-admissible.

Lemma 5.18

- $\mathrm{Adm}(A + B \mid y : S)$ *if* $\forall s \in S.\, (A + B)[s/y]$ type *and* $\mathrm{Adm}(A \mid y : S)$ *and* $\mathrm{Adm}(B \mid y : S)$.
- $\mathrm{Adm}(\Sigma x{:}A.B \mid y : S)$ *if* $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{Adm}(A \mid y : S)$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathrm{Adm}(\mathit{Void} \mid y : S)$, $\mathrm{Adm}(\mathit{Atom} \mid y : S)$, $\mathrm{Adm}(\mathbb{Z} \mid y : S)$ *and* $\mathrm{Adm}(\mathbb{E} \mid y : S)$

- $\mathrm{Adm}(a_1 = a_2\ in\ A \mid y : S)$ *if* $\forall s \in S. (a_1 = a_2\ in\ A)[s/y]$ type *and* $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a_1 \leq a_2 \mid y : S)$
- $\mathrm{Adm}(\overline{A} \mid y : S)$ *if* $\forall s \in S. \overline{A}[s/y]$ type *and* $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a\ in!\ A \mid y : S)$ *if* $\forall s \in S. (a\ in!\ A)[s/y]$ type

Predicate-admissibility of a function type is more complicated because a function argument with the type $A[e^{[\omega]}/x]$ does not necessarily belong to any of the finite approximations $A[e^{[j]}/x]$. To settle this, it is necessary to require a *coadmissibility* condition on the domain type. Then a function type will be predicate-admissible if the domain is weakly coadmissible and the codomain is predicate-admissible.

Definition 5.19 *A type T is* weakly coadmissible *for x in S (abbreviated* $\mathrm{WCoAdm}(T \mid x : S)$*) if:*

$$\begin{aligned}\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j. \forall k \geq j.\, e^{[k]} \in S) \wedge t = t' \in T[e^{[\omega]}/x] \Rightarrow& \\ (\exists j. \forall k \geq j.\, t = t' \in T[e^{[k]}/x])&\end{aligned}$$

A type T is coadmissible *for x in S (abbreviated* $\mathrm{CoAdm}(T \mid x : S)$*) if:*

$$\begin{aligned}\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j. \forall k \geq j.\, e^{[k]} \in S) \wedge t^{[\omega]} = t'^{[\omega]} \in T[e^{[\omega]}/x] \Rightarrow& \\ (\exists j. \forall k \geq j.\, t^{[k]} = t'^{[k]} \in T[e^{[k]}/x])&\end{aligned}$$

Lemma 5.20 $\mathrm{Adm}(\Pi x{:}A.B \mid y : S)$ *if* $\forall s \in S. (\Sigma x{:}A.B)[s/y]$ type *and* $\mathrm{WCoAdm}(A \mid y : S)$ *and* $\forall s \in S, a \in A[s/y]. \mathrm{Adm}(B[a/x] \mid y : S)$

Clearly coadmissibility implies weak coadmissibility. A general set of conditions listed in Lemma 5.21 establish weak and full coadmissibility for various types. Weak and full coadmissibility are closed under disjoint union and dependent sum formation, and full coadmissibility is additionally closed under equality-type formation. I use both notions of coadmissibility, rather than just adopting one or the other, because full coadmissibility is needed for equality types but under certain circumstances weak coadmissibility is easier to show (Proposition 5.22 below).

Lemma 5.21

- *$A + B$ is (weakly) coadmissible for y in S if* $\forall s \in S. (A + B)[s/y]$ type *and A and B are (weakly) coadmissible for y in S*
- $\mathrm{WCoAdm}(\Sigma x{:}A.B \mid y : S)$ *if* $\forall s \in S. (\Sigma x{:}A.B)[s/y]$ type *and* $\mathrm{WCoAdm}(A \mid y : S)$ *and* $\forall s \in S, a \in A[s/y]. \mathrm{WCoAdm}(B[a/x] \mid y : S)$
- $\mathrm{CoAdm}(\Sigma x{:}A.B \mid y : S)$ *if* $\forall s \in S. (\Sigma x{:}A.B)[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{CoAdm}(A \mid y : S)$ *and* $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- *Void, Atom, $\mathbb{Z}$ and $\mathbb{E}$ are strongly or weakly coadmissible for y in any S*
- $\mathrm{CoAdm}(a_1 = a_2\ in\ A \mid y : S)$ *if* $\forall s \in S. (a_1 = a_2\ in\ A)[s/y]$ type *and* $\mathrm{CoAdm}(A \mid y : S)$
- *$a_1 \leq a_2$ is strongly or weakly coadmissible for y in any S*
- *$\overline{A}$ is (weakly) coadmissible for y in S if* $\forall s \in S. \overline{A}[s/y]$ type *and A is (weakly) coadmissible for y is S*

- *a in! A is strongly or weakly coadmissible for y in S if* $\forall s \in S. (a \textit{ in! } A)[s/y]$ type

When T does not depend upon S, predicate-admissibility and weak coadmissibility become easier to show:

Proposition 5.22 *Suppose x does not appear free in T. Then:*

- $\mathrm{Adm}(T)$ *if* $\mathrm{Adm}(T \mid x : S)$ *and S is inhabited*
- $\mathrm{Adm}(T \mid x : S)$ *if* $\mathrm{Adm}(T)$
- $\mathrm{WCoAdm}(T \mid x : S)$

There remains one more result related to predicate-admissibility. Suppose one wishes to show $\mathrm{Adm}(T \mid x : S)$ where T depends upon x. There are two ways that x may be used in T. First, T might contain an equality type where x appears in one or both of the equands. In that case, predicate-admissibility can be shown with the tools discussed above. Second, T may be an expression that computes a type from x. In this case, T can be simplified using direct computation (Section 4.1.7), but another tool will be needed if T performs any case analysis (as in the embedding of the λ^K disjoint union type in Section 3.3.1).

Lemma 5.23 *Consider a type $case(d, x.A, x.B)$ that depends upon y from S. Suppose there exist T_1 and T_2 such that:*

- $\forall s \in S. d[s/y] \in (T_1 + T_2)[s/y]$
- $\forall s \in S, t \in T_1[s/y]. A[s,t/y,x]$ type
- $\forall s \in S, t \in T_2[s/y]. B[s,t/y,x]$ type
- $\Sigma y{:}S.T_1$ type *and* $\Sigma y{:}S.T_2$ type

Then the following are the case:

- $\mathrm{Adm}(case(d, x.A, x.B) \mid y : S)$ *if* $\mathrm{Adm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$
- $\mathrm{WCoAdm}(case(d, x.A, x.B) \mid y : S)$ *if* $\mathrm{WCoAdm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ *and* $\mathrm{WCoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$
- $\mathrm{CoAdm}(case(d, x.A, x.B) \mid y : S)$ *if* $\mathrm{CoAdm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ *and* $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$

5.3.2 Monotonicity

In some cases a very simple device may be used to show admissibility. We say that a type is monotone if it respects computational approximation, and it is easy to show that all monotone types are admissible.

Definition 5.24 *A type T is* monotone *(abbreviated* $\mathrm{Mono}(t)$*) if $t = t' \in T$ whenever $t \in T$ and $t \leq t'$.*

Lemma 5.25 *All monotone types are admissible.*

Proof

Let f, t and t' be arbitrary and suppose there exists j such that $t^{[j]} = t'^{[j]} \in T$. Since $t^{[j]} \leq t^{[\omega]}$ and $t'^{[j]} \leq t'^{[\omega]}$, it follows that $t^{[j]} = t^{[\omega]} \in T$ and $t'^{[j]} = t'^{[\omega]} \in T$. The result follows directly. □

All type constructors are monotone except universes and partial types, which are never monotone. The proof of this fact is easy [53].

Proposition 5.26

- $\mathrm{Mono}(A + B)$ *if* $\mathrm{Mono}(A)$ *and* $\mathrm{Mono}(B)$
- $\mathrm{Mono}(\Pi x{:}A.B)$ *if* $\mathrm{Mono}(A)$ *and* $\forall a \in A.\, \mathrm{Mono}(B[a/x])$
- $\mathrm{Mono}(\Sigma x{:}A.B)$ *if* $\mathrm{Mono}(A)$ *and* $\forall a \in A.\, \mathrm{Mono}(B[a/x])$
- $\mathrm{Mono}(\mathit{Void})$, $\mathrm{Mono}(\mathit{Atom})$, $\mathrm{Mono}(\mathbb{Z})$, $\mathrm{Mono}(\mathbb{E})$, $\mathrm{Mono}(a_1 = a_2 \in A)$, $\mathrm{Mono}(a_1 \leq a_2)$ *and* $\mathrm{Mono}(a\ \mathit{in!}\ A)$

5.3.3 Set and Quotient Types

Given the parallel between the dependent product and set types, it is natural to expect that set types would have a similar admissibility rule: that $\{x : A \mid B\}$ if A is admissible and B is predicate-admissible for x in A. Surprisingly, this turns out not to be the case. Suppose that $t^{[k]} \in \{x : A \mid B\}$ for all $k \geq j$. Then for every $k \geq j$, there exists some term $b_k \in B[t^{[k]}/x]$. We would like it to follow by predicate-admissibility that there exists $b_\omega \in B[t^{[\omega]}/x]$, but it does not. The problem is that each b_k can be a completely different term, and predicate-admissibility applies only when each b_k is of the form $b[f^k/w]$ for a single fixed b.

Intuitively, the desired rule fails because the set type $\{x : A|B\}$ suppresses the computational content of B and therefore B can be inhabited *non-uniformly,* by unrelated terms for related members of A. In contrast, if the chain $t^{[j]}, t^{[j+1]}, t^{[j+2]}, \ldots$ belongs to $\Sigma x{:}A.B$, then the chain $\pi_2(t)^{[j]}, \pi_2(t)^{[j+1]}, \pi_2(t)^{[j+2]}, \ldots$ uniformly inhabits B.

For a concrete example, consider:

$$\begin{array}{rcl} T & \stackrel{\text{def}}{=} & \{g : \mathbb{Z} \to \overline{\mathit{Unit}} \mid \exists n{:}\mathbb{Z}.\, \neg(gn\ \mathit{in!}\ \mathbb{Z})\} \\ f & \stackrel{\text{def}}{=} & \lambda h.\, \lambda x.\ \mathit{if}\ x \leq_Z 0\ \mathit{then}\ \star\ \mathit{else}\ h(x-1) \\ t & \stackrel{\text{def}}{=} & \lambda y.\, wy \end{array}$$

The type T is not admissible: For all k, $(t^{[k]_f})k$ diverges, so $t^{[k]_f} \in T$; but $t^{[\omega]_f}$ converges for all arguments, so $t^{[\omega]_f} \notin T$. However, $\exists n{:}\mathbb{Z}.\, \neg(gn\ \mathit{in!}\ \mathbb{Z})$ is predicate-admissible for g in $\mathbb{Z} \to \overline{\mathit{Unit}}$. The problem is that the inhabiting integers are not related by computational approximation; that is, they are not uniform.

To show a set type admissible, we need to be able to show that the selection predicate can be inhabited uniformly:

Lemma 5.27 *The type* $\{x : A \mid B\}$ *is admissible if:*

- $\mathrm{Adm}(A)$, *and*

- $\mathrm{Adm}(B \mid x : A)$, *and*
- *there exists* b *such that* $b[a/x] \in B[a/x]$ *whenever* $a \in A$ *and* $\exists b'.\, b' \in B[a/x]$.

Another way to state the uniformity condition (the third clause) is that the computational content of B should be recoverable from knowing it exists (*i.e.,* there exists a function with type $\Pi x{:}A.\, \downarrow B \to B$. In fact, the two versions are equivalent modulo a few functionality requirements:

Proposition 5.28

- *If* $f \in \Pi x{:}A.\, \downarrow B \to B$ *then* $b[a/x] \in B[a/x]$ *whenever* $a \in A$ *and* $\exists b'.\, b' \in B[a/x]$, *where* $b = f\, x\, \star$.
- *If* $b[a/x] = b[a'/x] \in B[a/x]$ *whenever* $a = a' \in A$ *and* $\exists b'.\, B[a/x]$ *and if* $B[a/x] = B[a'/x]$ *whenever* $a = a' \in A$ *then* $\lambda x. \lambda y.\, b \in \Pi x{:}A.\, \downarrow B \to B$.

We may give a similar predicate-admissibility condition:

Lemma 5.29 $\mathrm{Adm}(\{x : A \mid B\} \mid y : S)$ *if* $\forall s \in S.\, (\{x : A \mid B\})[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{Adm}(A \mid y : S)$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : \Sigma y{:}S.A)$, *and there exists* b *such that* $b[a, s/x, y] \in B[a, s/x, y]$ *whenever* $s \in S$ *and* $a \in A[s/y]$ *and* $\exists b'.\, b' \in B[a, s/x, y]$

Coadmissibility and monotonicity work on single terms, not chains, so the uniformity issue does not arise, resulting in conditions fairly similar to those for dependent products:

Lemma 5.30

- $\mathrm{WCoAdm}(\{x : A \mid B\} \mid y : S)$ *if* $\forall s \in S.\, \{x : A \mid B\}[s/y]$ type *and* $\mathrm{WCoAdm}(A \mid y : S)$ *and* $\forall s \in S, a \in A[s/y].\, \mathrm{WCoAdm}(B[a/x] \mid y : S)$
- $\mathrm{CoAdm}(\{x : A \mid B\} \mid y : S)$ *if* $\forall s \in S.\, \{x : A \mid B\}[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{CoAdm}(A \mid y : S)$ *and* $\mathrm{WCoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathrm{Mono}(\{x : A \mid B\})$ *if* $\mathrm{Mono}(A)$

The conditions for quotient types are similar to those for set types:

Lemma 5.31

- $\mathrm{Adm}(xy{:}A/\!/B)$ *if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/x, y] \mid z : A \times A)$, *and there exists* b *such that* $b[a, a'/x, y] \in B[a, a'/x, y]$ *whenever* $a \in A$ *and* $a' \in A$ *and* $\exists b'.\, b' \in B[a, a'/x, y]$.
- $\mathrm{Adm}(xy{:}A/\!/B \mid z : S)$ *if* $\forall s \in S.(xy{:}A/\!/B)[s/z]$ type *and* $\Sigma z{:}S.(A \times A)$ type *and* $\mathrm{Adm}(A \mid z : S)$ *and* $\mathrm{Adm}(B[\pi_1(z'), \pi_1(\pi_2(z')), \pi_2(\pi_2(z'))/z, x, y] \mid z' : \Sigma z{:}S.(A \times A))$, *and there exists* b *such that* $b[a, a', s/x, y, z] \in B[a, a', s/x, y, z]$ *whenever* $s \in S$ *and* $a \in A[s/z]$ *and* $a' \in A[s/z]$ *and* $\exists b'.\, b' \in B[a, a', s/x, y, z]$.
- $\mathrm{WCoAdm}(xy{:}A/\!/B \mid z : S)$ *if* $\forall s \in S.\, (xy{:}A/\!/B)[s/z]$ type *and* $\mathrm{WCoAdm}(A \mid z : S)$ *and* $\forall s \in S, a \in A[s/z], a' \in A[s/z].\, \mathrm{WCoAdm}(B[a, a'/x, y] \mid z : S)$
- $\mathrm{CoAdm}(xy{:}A/\!/B \mid z : S)$ *if* $\forall s \in S.\, (xy{:}A/\!/B)[s/z]$ type *and* $\Sigma z{:}S.(A \times A)$ type *and* $\mathrm{CoAdm}(A \mid z : S)$ *and* $\mathrm{CoAdm}(B[\pi_1(z'), \pi_1(\pi_2(z')), \pi_2(\pi_2(z'))/z, x, y] \mid z' : \Sigma z{:}S.(A \times A))$
- $\mathrm{Mono}(xy{:}A/\!/B)$ *if* $\mathrm{Mono}(A)$

5.3.4 Summary

Figure 5.1 provides a summary of the basic admissibility results of this chapter. It is worthwhile to note that all these results are proved constructively, with the exception of (weak and full) coadmissibility of partial types. The following theorem shows that the proofs of coadmissibility of partial types are necessarily classical; if a constructive proof existed then one could extract an algorithm meeting the theorem's specification, which can be used to solve the halting problem.

Theorem 5.32 *There does not exist an algorithm that computes an integer j such that $\forall k \geq j.\, t = t' \in \overline{T}[e^{[k]}/x]$, when given S, T, f, t, t', e and i such that:*

- $\forall s \in S.\, \overline{T}[s/x]$ type
- $\mathrm{CoAdm}(T \mid x : S)$
- $e^{[\omega]} \in S$
- $\forall k \geq i.\, e^{[k]} \in S$
- $t = t' \in \overline{T}[e^{[\omega]}/x]$

Recall the inadmissible type T from Theorem 5.1. That type fails the predicate-admissibility condition because of the negative appearance of a function type, which could not be shown weakly coadmissible, and it fails the monotonicity condition because it contains the partial type $\overline{\mathbb{N}}$.

5.4 Conclusions

An interesting avenue for future investigation would be to find some negative results characterizing inadmissible types. Such negative results would be particularly interesting if they could be given a syntactic character, like the results of this chapter. Along these lines, it would be interesting to find whether the inability to show coadmissibility of function types represents a weakness of this proof technique or an inherent limitation.

The results presented above provide *metatheoretical* justification for the fixpoint principle over many types. In order for these results to be useful in theorem proving, they must be introduced into the logic. One way to do this, and the way it is done here, is to introduce types to represent the assertions $\mathrm{Adm}(T)$, $\mathrm{Adm}(T \mid x : S)$, etc., that are inhabited exactly when the underlying assertion is true (in much that same way as the equality type is inhabited exactly when the equands are equal), and to add rules relating to these types that correspond to the lemmas of Section 5.3. This brings the tools into the system in a semantically justifiable way, but it is unpleasant in that it leads to a proliferation of new types and inference rules stemming from discoveries outside the logic. Of the Nuprl proof rules of Appendix B, 84 rules (not quite half) deal with admissibility. It would be preferable to deal with admissibility within the logic. A theory with intensional reasoning principles, such as the one proposed in Constable and Crary [23], would allow reasoning about computation internally. Then these results could be proved within the theory and the only extra rule that would be required would be a single rule relating admissibility to the the fixpoint principle.

However they are placed into the logic, these results allow for recursive computation on a wide variety of types. This make partial types and fixpoint induction a useful tool in type-theoretic theorem provers. It also makes it possible to study many recursive programs that used to be barred from the logic because they could not be typed.

	For $T \equiv$		
	$A + B$	$\Pi x{:}A.B$	$\Sigma x{:}A.B$
$\mathrm{Adm}(T)$ if T type and	$\mathrm{Adm}(A) \wedge \mathrm{Adm}(B)$	$\forall a \in A.\, \mathrm{Adm}(B[a/x])$	$\mathrm{Adm}(A) \wedge \mathrm{Adm}(B \mid x : A)$
$\mathrm{Adm}(T \mid y : S)$ if $\forall s \in S. T[s/y]$ type and	$\mathrm{Adm}(A \mid y : S) \wedge$ $\mathrm{Adm}(B \mid y : S)$	$\mathrm{WCoAdm}(A \mid y : S) \wedge$ $\forall s \in S, a \in A[s/y].$ $\mathrm{Adm}(B[a/x] \mid y : S)$	$\mathrm{Adm}(A \mid y : S) \wedge$ $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x]$ $\mid z : (\Sigma y : S.A)) \wedge$ $\Sigma y{:}S.A$ type
$\mathrm{WCoAdm}(T \mid y : S)$ if $\forall s \in S. T[s/y]$ type and	$\mathrm{WCoAdm}(A \mid y : S) \wedge$ $\mathrm{WCoAdm}(B \mid y : S)$	—	$\mathrm{WCoAdm}(A \mid y : S) \wedge$ $\forall s \in S, a \in A[s/y].$ $\mathrm{WCoAdm}(B[a/x] \mid y : S)$
$\mathrm{CoAdm}(T \mid y : S)$ if $\forall s \in S. T[s/y]$ type and	$\mathrm{CoAdm}(A \mid y : S) \wedge$ $\mathrm{CoAdm}(B \mid y : S)$	—	$\mathrm{CoAdm}(A \mid y : S) \wedge$ $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x]$ $\mid z : (\Sigma y : S.A)) \wedge$ $\Sigma y{:}S.A$ type
$\mathrm{Mono}(T)$ if	$\mathrm{Mono}(A) \wedge \mathrm{Mono}(B)$	$\mathrm{Mono}(A) \wedge$ $\forall a \in A.\, \mathrm{Mono}(B[a/x])$	$\mathrm{Mono}(A) \wedge$ $\forall a \in A.\, \mathrm{Mono}(B[a/x])$

	For $T \equiv$		
	$\mathit{Void}, \mathit{Atom}, \mathbb{Z}, \mathbb{E}, a_1 \leq a_2, a\ \mathit{in!}\ A$	$a_1 = a_2\ \mathit{in}\ A$	$\overline{A}$
$\mathrm{Adm}(T)$ if T type and	yes	yes	$\mathrm{Adm}(A)$
$\mathrm{Adm}(T \mid y : S)$ if $\forall s \in S. T[s/y]$ type and	yes	$\mathrm{Adm}(A \mid y : S)$	$\mathrm{Adm}(A \mid y : S)$
$\mathrm{WCoAdm}(T \mid y : S)$ if $\forall s \in S. T[s/y]$ type and	yes	$\mathrm{CoAdm}(A \mid y : S)$	$\mathrm{WCoAdm}(A \mid y : S)$
$\mathrm{CoAdm}(T \mid y : S)$ if $\forall s \in S. T[s/y]$ type and	yes	$\mathrm{CoAdm}(A \mid y : S)$	$\mathrm{CoAdm}(A \mid y : S)$
$\mathrm{Mono}(T)$ if	yes	yes	—

Figure 5.1: Admissibility, coadmissibility and monotonicity conditions

Chapter 6

Conclusion

In this dissertation I have explored how to draw formal connections between type theory and practical programming languages. These connections allow the full power of foundational type theory to be brought to bear on research problems arising from *real* programs and programming languages.

The least consequence of these contributions is that program verification results in type-theoretic systems may be directly applied to real programs, without trusting an intermediate hand translation into type theory. More significantly, type theory may be applied to programming languages as a whole; this allows us to understand programming languages within a rich mathematical framework, but one that is inherently computational and that also does not disturb the essential structure of programs.

Three main technical achievements made this possible. First, I presented a semantics for a practical programming calculus in the framework of the Nuprl type theory. This semantics was stated as a syntax-directed embedding of the constructs, and used novel type-theoretic mechanisms to address the language mechanisms of recursion, a rich dependent kind structure, and a higher-order modules system with translucent signatures.

Second, backing up the type-theoretic semantics was the first constructive type theory to include an intrinsic notion of equality, and to support reasoning about partial and total functions. All three are necessary to allow interesting reasoning about real programs. Partiality is necessary because real programs often make use of recursion. However, a notion of totality is also necessary to allow reasoning about *pure values,* which are central to real programs as well.

Third, broad new techniques for showing admissibility of types for fixpoint induction were necessary for each of the previous contributions. Although the techniques I present are still conservative, they make possible admissibility proofs for many types that could not be shown admissible by previous techniques. For example, types arising in the embedding of disjoint unions (Section 3.3.1) and in fixpoint induction (Section 4.3.3) required the new techniques to be shown admissible.

The broadest aim of this work has been to bring theory and practice closer together. Aside from the specific contributions that make up this dissertation, I have found that an important way this work has helped achieve that is as a conceptual tool for thinking about programming and programming languages. Practical languages are made up of diverse sets of often complicated mechanisms. In contrast, languages for theoretical study usually are made up of small sets of elegant mechanisms. Type theory, like other theoretical languages, is made up of elegant mechanisms, but I have shown in this dissertation that those mechanisms are sufficiently powerful to account for most of the diverse mechanisms of practical languages, and those that remain show great promise of being addressed as well (recall Section 3.5).

Consequently, I have found it very profitable to use type theory as a tool for considering general programming issues. The power and elegance of type theory allow a researcher to derive elegant solutions to diverse problems but do not require one to ignore practical issues to get them.

Appendix A

Lambda-K Typing Rules

Level Definitions

$$
\begin{array}{lcl}
\mathit{level}(\mathit{Type}_i) & \stackrel{\mathrm{def}}{=} & i \\
\mathit{level}(\mathcal{P}_i(c)) & \stackrel{\mathrm{def}}{=} & i \\
\mathit{level}(\mathcal{S}_i(c)) & \stackrel{\mathrm{def}}{=} & i \\
\mathit{level}(\Pi\alpha{:}\kappa_1.\kappa_2) & \stackrel{\mathrm{def}}{=} & \max(\mathit{level}(\kappa_1), \mathit{level}(\kappa_2)) \\
\mathit{level}(\{\ell_i \triangleright \alpha_i : \kappa_i{}^{[i=1\ldots n]}\}) & \stackrel{\mathrm{def}}{=} & \max_{i=1}^{n} \mathit{level}(\kappa_i)
\end{array}
$$

Kind Formation and Equality

$$\frac{\Gamma \vdash_K \kappa_2 = \kappa_1}{\Gamma \vdash_K \kappa_1 = \kappa_2} \quad \text{(KindEqSymm)}$$

$$\frac{\Gamma \vdash_K \kappa_1 = \kappa_2 \quad \Gamma \vdash_{IL} \kappa_2 = \kappa_3}{\Gamma \vdash_K \kappa_1 = \kappa_3} \quad \text{(KindEqTrans)}$$

$$\frac{}{\Gamma \vdash_K \mathit{Type}_i = \mathit{Type}_i} \quad \text{(Type)}$$

$$\frac{\Gamma \vdash_K \kappa_1 = \kappa_1' \quad \Gamma[\alpha : \kappa_1] \vdash_K \kappa_2 = \kappa_2'}{\Gamma \vdash_K \Pi\alpha{:}\kappa_1.\kappa_2 = \Pi\alpha{:}\kappa_1'.\kappa_2'} \ (\alpha \notin \Gamma) \quad \text{(Pi)}$$

$$\frac{\Gamma[\alpha_j : \kappa_j]^{[j=1\ldots i-1]} \vdash_K \kappa_i = \kappa_i' \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash_K \{\ell_i \triangleright \alpha_i : \kappa_i{}^{[i=1\ldots n]}\} = \{\ell_i \triangleright \alpha_i : \kappa_i'{}^{[i=1\ldots n]}\}} \ (\alpha_i^{[i=1\ldots m]} \notin \Gamma) \quad \text{(DepRecord)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i}{\Gamma \vdash_K \mathcal{P}_i(c_1) = \mathcal{P}_i(c_2)} \quad \text{(Pow)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i}{\Gamma \vdash_K \mathcal{S}_i(c_1) = \mathcal{S}_i(c_2)} \quad \text{(Sing)}$$

Subkinding

$$\frac{\Gamma \vdash_K \kappa_1 = \kappa_2}{\Gamma \vdash_K \kappa_1 \preceq \kappa_2} \quad \text{(SubkindReflex)}$$

$$\frac{\Gamma \vdash_K \kappa_1 \preceq \kappa_2 \quad \Gamma \vdash_{IL} \kappa_2 \preceq \kappa_3}{\Gamma \vdash_K \kappa_1 \preceq \kappa_3} \quad \text{(SubkindTrans)}$$

$$\frac{}{\Gamma \vdash_K \mathit{Type}_i \preceq \mathit{Type}_j} \ (i \leq j) \quad \text{(TypeSub)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K \kappa_1' \preceq \kappa_1 \\ \Gamma[\alpha : \kappa_1'] \vdash_K \kappa_2 \preceq \kappa_2' \\ \Gamma[\alpha : \kappa_1] \vdash_K \kappa_2 \ \text{kind}\end{array}}{\Gamma \vdash_K \Pi\alpha{:}\kappa_1.\kappa_2 \preceq \Pi\alpha{:}\kappa_1'.\kappa_2'} \ (\alpha \notin \Gamma) \quad \text{(PiSub)}$$

$$\frac{\begin{array}{ll}\Gamma[\alpha_j : \kappa_j]^{[j=1\ldots i-1]} \vdash_K \kappa_i \preceq \kappa_i' & \text{for } 1 \leq i \leq n \\ \Gamma[\alpha_j : \kappa_j]^{[j=1\ldots i-1]} \vdash_K \kappa_i \ \text{kind} & \text{for } n < i \leq m \\ \Gamma[\alpha_j : \kappa_j']^{[j=1\ldots i-1]} \vdash_K \kappa_i' \ \text{kind} & \text{for } 1 \leq i \leq n\end{array}}{\Gamma \vdash_K \{\ell_i \triangleright \alpha_i : \kappa_i{}^{[i=1\ldots m]}\} \preceq \{\ell_i \triangleright \alpha_i : \kappa_i'{}^{[i=1\ldots n]}\}} \ (\alpha_i^{[i=1\ldots m]} \notin \Gamma, m \geq n) \quad \text{(DepRecordSub)}$$

$$\frac{\Gamma \vdash_K c_1 \preceq c_2}{\Gamma \vdash_K \mathcal{P}_i(c_1) \preceq \mathcal{P}_j(c_2)} \ (i \leq j) \quad \text{(PowSub)}$$

$$\frac{\Gamma \vdash_K c : \mathit{Type}_i}{\Gamma \vdash_K \mathcal{P}_i(c) \preceq \mathit{Type}_i} \quad \text{(PowType)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i}{\Gamma \vdash_K \mathcal{S}_i(c_1) \preceq \mathcal{S}_j(c_2)} \ (i \leq j) \quad \text{(SingSub)}$$

$$\frac{\Gamma \vdash_K c : \mathit{Type}_i}{\Gamma \vdash_K \mathcal{S}_i(c) \preceq \mathcal{P}_i(c)} \quad \text{(SingPow)}$$

Constructor Formation and Equality

$$\frac{\Gamma \vdash_K c_2 = c_1 : \kappa}{\Gamma \vdash_K c_1 = c_2 : \kappa} \quad \text{(EqSymm)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \kappa \quad \Gamma \vdash_K c_2 = c_3 : \kappa}{\Gamma \vdash_K c_1 = c_3 : \kappa} \quad \text{(EqTrans)}$$

$$\frac{}{\Gamma \vdash_K \alpha = \alpha : \kappa} \ (\Gamma(\alpha) = \kappa) \quad \text{(TypeVar)}$$

$$\frac{\Gamma \vdash_K \kappa_1 \ \text{kind} \quad \Gamma[\alpha : \kappa_1] \vdash_K c = c' : \kappa_2}{\Gamma \vdash_K \lambda\alpha{:}\kappa_1.c = \lambda\alpha{:}\kappa_1.c' : \Pi\alpha{:}\kappa_1.\kappa_2} \ (\alpha \notin \Gamma) \quad \text{(PiIntro)}$$

$$\frac{\Gamma \vdash_K c_1 = c_1' : \Pi\alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash_K c_2 = c_2' : \kappa_1}{\Gamma \vdash_K c_1[c_2] = c_1'[c_2'] : \kappa_2[c_2/\alpha]} \quad \text{(PiElim)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K c_i = c_i' : \kappa_i[c_j^{[j=1\ldots i-1]}/\alpha_j^{[j=1\ldots i-1]}] \\ \text{(for } 1 \leq i \leq n) \\ \Gamma \vdash_K \{\ell_i \triangleright \alpha_i : {\kappa_i}^{[i=1\ldots n]}\} \ \text{kind}\end{array}}{\Gamma \vdash_K \{\ell_i = {c_i}^{[i=1\ldots n]}\} = \{\ell_i = {c_i'}^{[i=1\ldots n]}\} : \{\ell_i \triangleright \alpha_i : {\kappa_i}^{[i=1\ldots n]}\}} \quad \text{(DepRecordIntro)}$$

$$\frac{\Gamma \vdash_K c = c' : \{\ell_j \triangleright \alpha_j : {\kappa_j}^{[i=1\ldots n]}\}}{\Gamma \vdash_K \pi_{\ell_i}(c) = \pi_{\ell_i}(c') : \kappa_i[\pi_{\ell_j}(c)^{[j=1\ldots i-1]}/\alpha_j^{[j=1\ldots i-1]}]} \ (1 \leq i \leq n) \quad \text{(DepRecordElim)}$$

$$\frac{\Gamma \vdash_K c_1 = c_1' : \mathit{Type}_i \quad \Gamma \vdash_K c_2 = c_2' : \mathit{Type}_i}{\Gamma \vdash_K c_1 \rightarrow c_2 = c_1' \rightarrow c_2' : \mathit{Type}_i} \quad \text{(Arrow)}$$

$$\frac{\Gamma \vdash_K c_1 = c_1' : \mathit{Type}_i \quad \Gamma \vdash_K c_2 = c_2' : \mathit{Type}_i}{\Gamma \vdash_K c_1 \Rightarrow c_2 = c_1' \Rightarrow c_2' : \mathit{Type}_i} \quad \text{(TArrow)}$$

$$\frac{\Gamma \vdash_K \kappa_1 = \kappa_2 \quad \Gamma[\alpha : \kappa_1] \vdash_K c = c' : \mathit{Type}_i}{\Gamma \vdash_K \forall\alpha{:}\kappa_1.c_1 = \forall\alpha{:}\kappa_2.c_2 : \mathit{Type}_i} \ (\alpha \notin \Gamma, \mathit{level}(\kappa_1) < i) \quad \text{(Quant)}$$

$$\frac{\Gamma \vdash_K c_i = c_i' : \mathit{Type}_j \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash_K \{\ell_1 : c_1, \ldots, \ell_n : c_n\} = \{\ell_1 : c_1', \ldots, \ell_n : c_n'\} : \mathit{Type}_j} \quad \text{(Record)}$$

$$\frac{\Gamma \vdash_K c_i = c_i' : \mathit{Type}_j \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash_K \langle\ell_1 : c_1, \ldots, \ell_n : c_n\rangle = \langle\ell_1 : c_1', \ldots, \ell_n : c_n'\rangle : \mathit{Type}_j} \quad \text{(Union)}$$

$$\frac{\Gamma \vdash_K c = c' : \kappa_1 \quad \Gamma \vdash_K \kappa_1 \preceq \kappa_2}{\Gamma \vdash_K c = c' : \kappa_2} \quad \text{(Subkind)}$$

$$\frac{\Gamma \vdash_K (\lambda\alpha{:}\kappa'.c_1)[c_2] : \kappa}{\Gamma \vdash_K (\lambda\alpha{:}\kappa'.c_1)[c_2] = c_1[c_2/\alpha] : \kappa} \quad \text{(PiBeta)}$$

$$\frac{\Gamma \vdash_K \lambda\alpha{:}\kappa_1'.c[\alpha] : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash_K \lambda\alpha{:}\kappa_1'.c[\alpha] = c : \Pi\alpha{:}\kappa_1.\kappa_2} \ (\alpha \notin c) \quad \text{(PiEta)}$$

$$\frac{\Gamma \vdash_K \pi_{\ell_j}(\{\ell_i = {c_i}^{[i=1\ldots n]}\}) : \kappa}{\Gamma \vdash_K \pi_{\ell_j}(\{\ell_i = {c_i}^{[i=1\ldots n]}\}) = c_j : \kappa} \ (1 \leq j \leq n) \quad \text{(DepRecordBeta)}$$

$$\frac{\Gamma \vdash_K \{\ell_i = \pi_{\ell_i}(c)^{[i=1\ldots n]}\} : \{\ell_i \triangleright \alpha_i : {\kappa_i}^{[i=1\ldots n]}\}}{\Gamma \vdash_K \{\ell_i = \pi_{\ell_i}(c)^{[i=1\ldots n]}\} = c : \{\ell_i \triangleright \alpha_i : {\kappa_i}^{[i=1\ldots n]}\}} \quad \text{(DepRecordEta)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i}{\Gamma \vdash_K c_1 : \mathcal{S}_i(c_2)} \quad \text{(SingIntro)}$$

$$\frac{\Gamma \vdash_K c_1 : \mathcal{S}_i(c_2)}{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i} \quad \text{(SingElim)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i \quad \Gamma \vdash_K c_1 : \mathcal{P}_i(c_3)}{\Gamma \vdash_K c_1 = c_2 : \mathcal{P}_i(c_3)} \quad \text{(PowFun)}$$

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i \quad \Gamma \vdash_K c_1 : \mathcal{S}_i(c_3)}{\Gamma \vdash_K c_1 = c_2 : \mathcal{S}_i(c_3)} \quad \text{(SingFun)}$$

$$\frac{\Gamma \vdash_K m : \langle\kappa\rangle}{\Gamma \vdash_K \mathit{ext}(m) : \kappa} \quad \text{(TypeModElim)}$$

Subtyping

$$\frac{\Gamma \vdash_K c_1 \preceq c_2 \quad \Gamma \vdash_K c_2 \preceq c_3}{\Gamma \vdash_K c_1 \preceq c_3} \qquad \text{(SubTrans)}$$

$$\frac{\Gamma \vdash_K c_1' \preceq c_1 \quad \Gamma \vdash_K c_2 \preceq c_2'}{\Gamma \vdash_K c_1 \rightarrow c_2 \preceq c_1' \rightarrow c_2'} \qquad \text{(ArrowSub)}$$

$$\frac{\Gamma \vdash_K c_1' \preceq c_1 \quad \Gamma \vdash_K c_2 \preceq c_2'}{\Gamma \vdash_K c_1 \Rightarrow c_2 \preceq c_1' \Rightarrow c_2'} \qquad \text{(TArrowSub)}$$

$$\frac{\Gamma \vdash_K c_1 \ \text{type} \quad \Gamma \vdash_K c_2 \ \text{type}}{\Gamma \vdash_K c_1 \Rightarrow c_2 \preceq c_1 \rightarrow c_2} \qquad \text{(TotalSub)}$$

$$\frac{\begin{array}{l}\Gamma \vdash_K \kappa_2 \preceq \kappa_1 \\ \Gamma[\alpha : \kappa_2] \vdash_K c_1 \preceq c_2 \\ \Gamma[\alpha : \kappa_1] \vdash_K c_1 \ \text{type}\end{array}}{\Gamma \vdash_K \forall\alpha{:}\kappa_1.c_1 \preceq \forall\alpha{:}\kappa_2.c_2} \ (\alpha \notin \Gamma) \qquad \text{(QuantSub)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K c_i \preceq c_i' \quad \text{for } 1 \leq i \leq n \\ \Gamma \vdash_K c_i \ \text{type} \quad \text{for } n < i \leq m\end{array}}{\begin{array}{r}\Gamma \vdash_K \{\ell_1 : c_1, \ldots, \ell_n : c_m\} \preceq \\ \{\ell_1 : c_1', \ldots, \ell_n : c_n'\}\end{array}} \ (m \geq n) \qquad \text{(RecordSub)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K c_i \preceq c_i' \quad \text{for } 1 \leq i \leq m \\ \Gamma \vdash_K c_i' \ \text{type} \quad \text{for } m < i \leq n\end{array}}{\begin{array}{r}\Gamma \vdash_K \langle \ell_1 : c_1, \ldots, \ell_n : c_m \rangle \preceq \\ \langle \ell_1 : c_1', \ldots, \ell_n : c_n' \rangle\end{array}} \ (m \leq n) \qquad \text{(UnionSub)}$$

$$\frac{\Gamma \vdash_K c_1 \preceq c_2 \quad \Gamma \vdash_K c_1 : \mathit{Type}_i \quad \Gamma \vdash_K c_2 : \mathit{Type}_i}{\Gamma \vdash_K c_1 : \mathcal{P}_i(c_2)} \qquad \text{(PowIntro)}$$

$$\frac{\Gamma \vdash_K c_1 : \mathcal{P}_i(c_2)}{\Gamma \vdash_K c_1 \preceq c_2} \qquad \text{(PowElim)}$$

Typing

$$\frac{}{\Gamma \vdash_K x : c} \ (\Gamma(x) = c) \qquad \text{(Var)}$$

$$\frac{\Gamma \vdash_K c_1 \ \text{type} \quad \Gamma[x : c_1] \vdash_K e : c_2}{\Gamma \vdash_K \lambda x{:}c_1.e : c_1 \rightarrow c_2} \ (x \notin \Gamma) \qquad \text{(ArrowIntro)}$$

$$\frac{\Gamma \vdash_K e_1 : c_1 \rightarrow c_2 \quad \Gamma \vdash_K e_2 : c_1}{\Gamma \vdash_K e_1 e_2 : c_2} \qquad \text{(ArrowElim)}$$

$$\frac{\Gamma \vdash_K c_1 \ \text{type} \quad \Gamma[x : c_1] \vdash_K e \downarrow c_2}{\Gamma \vdash_K \lambda x{:}c_1.e : c_1 \Rightarrow c_2} \ (x \notin \Gamma) \qquad \text{(TArrowIntro)}$$

$$\frac{\Gamma \vdash_K e_1 : c_1 \Rightarrow c_2 \quad \Gamma \vdash_K e_2 : c_1}{\Gamma \vdash_K e_1 e_2 : c_2} \qquad \text{(TArrowElim)}$$

$$\frac{\Gamma \vdash_K \kappa \ \text{kind} \quad \Gamma[\alpha : \kappa] \vdash_K e \downarrow c}{\Gamma \vdash_K \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.c} \ (\alpha \notin \Gamma) \qquad \text{(QuantIntro)}$$

$$\frac{\Gamma \vdash_K e : \forall\alpha{:}\kappa.c_2 \quad \Gamma \vdash_K c_1 : \kappa}{\Gamma \vdash_K e[c_1] : c_2[c_1/\alpha]} \qquad \text{(QuantElim)}$$

$$\frac{\Gamma \vdash_K e_i : c_i \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash_K \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : c_1, \ldots, \ell_n : c_n\}} \qquad \text{(RecordIntro)}$$

$$\frac{\Gamma \vdash_K e : \{\ell_1 : c_1, \ldots, \ell_n : c_n\}}{\Gamma \vdash_K \pi_{\ell_i}(e) : c_i} \ (1 \leq i \leq n) \qquad \text{(RecordElim)}$$

$$\frac{\Gamma \vdash_K e : \tau}{\Gamma \vdash_K \mathit{inj}_\ell(e) : \langle \ell : \tau \rangle} \qquad \text{(UnionIntro)}$$

$$\frac{\begin{array}{l}\Gamma \vdash_K e : \langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle \\ \Gamma[x_i : \tau_i] \vdash_K e_i : \tau \quad (\text{for } 1 \leq i \leq n)\end{array}}{\Gamma \vdash_K \mathit{case}(e, \ell_1 \triangleright x_1.e_1, \ldots, \ell_n \triangleright x_n.e_n) : \tau} \qquad \text{(UnionElim)}$$

$$\frac{\Gamma \vdash_K e : (c_1 \rightarrow c_2) \rightarrow (c_1 \rightarrow c_2)}{\Gamma \vdash_K \mathit{fix}_{c_1 \rightarrow c_2}(e) : c_1 \rightarrow c_2} \qquad \text{(Fix)}$$

$$\frac{\Gamma \vdash_K m : \langle\!\langle c \rangle\!\rangle}{\Gamma \vdash_K \mathit{ext}(m) : c} \qquad \text{(TermModElim)}$$

$$\frac{\Gamma \vdash_K e : c_1 \quad \Gamma \vdash_K c_1 \preceq c_2}{\Gamma \vdash_K e : c_2} \qquad \text{(Subtype)}$$

Valuability

$$\frac{}{\Gamma \vdash_K x \downarrow c} \ (\Gamma(x) = c) \qquad \text{(VarHalt)}$$

$$\frac{\Gamma \vdash_K c_1 \text{ type} \quad \Gamma[x : c_1] \vdash_K e : c_2}{\Gamma \vdash_K \lambda x{:}c_1.e \downarrow c_1 \to c_2} \ (x \notin \Gamma) \qquad \text{(ArrowIntroHalt)}$$

$$\frac{\Gamma \vdash_K c_1 \text{ type} \quad \Gamma[x : c_1] \vdash_K e \downarrow c_2}{\Gamma \vdash_K \lambda x{:}c_1.e \downarrow c_1 \Rightarrow c_2} \ (x \notin \Gamma) \qquad \text{(TArrowIntroHalt)}$$

$$\frac{\Gamma \vdash_K e_1 \downarrow c_1 \Rightarrow c_2 \quad \Gamma \vdash_K e_2 \downarrow c_1}{\Gamma \vdash_K e_1 e_2 \downarrow c_2} \qquad \text{(TArrowElimHalt)}$$

$$\frac{\Gamma \vdash_K \kappa \text{ kind} \quad \Gamma[\alpha : \kappa] \vdash_K e \downarrow c}{\Gamma \vdash_K \Lambda\alpha{:}\kappa.e \downarrow \forall\alpha{:}\kappa.c} \ (\alpha \notin \Gamma) \qquad \text{(QuantIntroHalt)}$$

$$\frac{\Gamma \vdash_K e \downarrow \forall\alpha{:}\kappa.c_2 \quad \Gamma \vdash_K c_1 : \kappa}{\Gamma \vdash_K e[c_1] \downarrow c_2[c_1/\alpha]} \qquad \text{(QuantElimHalt)}$$

$$\frac{\Gamma \vdash_K e_i \downarrow c_i \quad \text{for } 1 \leq i \leq n}{\Gamma \vdash_K \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \downarrow \{\ell_1 : c_1, \ldots, \ell_n : c_n\}} \qquad \text{(RecordIntroHalt)}$$

$$\frac{\Gamma \vdash_K e \downarrow \{\ell_1 : c_1, \ldots, \ell_n : c_n\}}{\Gamma \vdash_K \pi_{\ell_i}(e) \downarrow c_i} \ (i \leq i \leq n) \qquad \text{(RecordElimHalt)}$$

$$\frac{\Gamma \vdash_K e \downarrow \tau}{\Gamma \vdash_K \mathit{inj}_\ell(e) \downarrow \langle \ell : \tau \rangle} \qquad \text{(UnionIntroHalt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K e \downarrow \langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle \\ \Gamma[x_i : \tau_i] \vdash_K e_i \downarrow \tau \quad (\text{for } 1 \leq i \leq n)\end{array}}{\Gamma \vdash_K \mathit{case}(e, \ell_1 \triangleright x_1.e_1, \ldots, \ell_n \triangleright x_n.e_n) \downarrow \tau} \qquad \text{(UnionElimHalt)}$$

$$\frac{\Gamma \vdash_K e \downarrow (c_1 \to c_2) \Rightarrow (c_1 \to c_2)}{\Gamma \vdash_K \mathit{fix}_{c_1 \to c_2}(e) \downarrow c_1 \to c_2} \qquad \text{(FixHalt)}$$

$$\frac{\Gamma \vdash_K m \downarrow \langle\!\langle c \rangle\!\rangle}{\Gamma \vdash_K \mathit{ext}(m) \downarrow c} \qquad \text{(TermModElimHalt)}$$

$$\frac{\Gamma \vdash_K e \downarrow c_1 \quad \Gamma \vdash_K c_1 \preceq c_2}{\Gamma \vdash_K e \downarrow c_2} \qquad \text{(SubtypeHalt)}$$

Signature Formation and Subsignatures

$$\frac{\Gamma \vdash_K \sigma_1 \preceq \sigma_2 \quad \Gamma \vdash_K \sigma_2 \preceq \sigma_3}{\Gamma \vdash_K \sigma_1 \preceq \sigma_3} \qquad \text{(SubsigTrans)}$$

$$\frac{\Gamma \vdash_K \kappa_1 \preceq \kappa_2}{\Gamma \vdash_K \langle \kappa_1 \rangle \preceq \langle \kappa_2 \rangle} \qquad \text{(TypeSig)}$$

$$\frac{\Gamma \vdash_K c_1 \preceq c_2}{\Gamma \vdash_K \langle\!\langle c_1 \rangle\!\rangle \preceq \langle\!\langle c_2 \rangle\!\rangle} \qquad \text{(TermSig)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K \sigma_1' \preceq \sigma_1 \\ \Gamma[s : \sigma_1'] \vdash_K \sigma_2 \preceq \sigma_2' \\ \Gamma[s : \sigma_1] \vdash_K \sigma_2 \text{ sig}\end{array}}{\Gamma \vdash_K \Pi s{:}\sigma_1.\sigma_2 \preceq \Pi s{:}\sigma_1'.\sigma_2'} \qquad \text{(PiSig)}$$

$$\frac{\begin{array}{c}\Gamma[s_j : \sigma_j]^{[j=1\ldots i-1]} \vdash_K \sigma_i \preceq \sigma_i' \quad \text{for } 1 \leq i \leq n \\ \Gamma[s_j : \sigma_j]^{[j=1\ldots i-1]} \vdash_K \sigma_i \text{ sig} \quad \text{for } n < i \leq m \\ \Gamma[s_j : \sigma_j']^{[j=1\ldots i-1]} \vdash_K \sigma_i' \text{ sig} \quad \text{for } 1 \leq i \leq n\end{array}}{\Gamma \vdash_K \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots m]}\} \preceq \{\ell_i \triangleright s_i : \sigma_i'{}^{[i=1\ldots n]}\}} \ (s_i^{[i=1\ldots m]} \notin \Gamma, m \geq n) \qquad \text{(RecordSig)}$$

Module Formation

$$\frac{}{\Gamma \vdash_K s : \sigma} \ (\Gamma(s) = \sigma) \qquad \text{(ModVar)}$$

$$\frac{\Gamma \vdash_K c : \kappa}{\Gamma \vdash_K \langle c \rangle : \langle \kappa \rangle} \qquad \text{(TypeMod)}$$

$$\frac{\Gamma \vdash_K e : c}{\Gamma \vdash_K \langle\!\langle e \rangle\!\rangle : \langle\!\langle c \rangle\!\rangle} \qquad \text{(TermMod)}$$

$$\frac{\Gamma \vdash_K \sigma \text{ sig} \quad \Gamma[s : \sigma_1] \vdash_K m : \sigma_2}{\Gamma \vdash_K \lambda s{:}\sigma_1.m : \Pi s{:}\sigma_1.\sigma_2} \qquad \text{(PiModIntro)}$$

$$\frac{\Gamma \vdash_K m_1 : \Pi s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_K m_2 : \sigma_1}{\Gamma \vdash_K m_1 m_2 : \sigma_2[m_2/s]} \qquad \text{(PiModElim)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K m_i : \sigma_i[m_j^{[j=1\ldots i-1]}/s_j^{[j=1\ldots i-1]}] \\ (\text{for } 1 \leq i \leq n) \\ \Gamma \vdash_K \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\} \text{ sig}\end{array}}{\Gamma \vdash_K \{\ell_i = m_i{}^{[i=1\ldots n]}\} : \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\}} \qquad \text{(RecordModIntro)}$$

$$\frac{\Gamma \vdash_K m : \{\ell_j \triangleright s_j : \sigma_j{}^{[i=1\ldots n]}\}}{\Gamma \vdash_K \pi_{\ell_i}(m) : \sigma_i[\pi_{\ell_j}(m)^{[j=1\ldots i-1]}/s_j^{[j=1\ldots i-1]}]} \quad (1 \leq i \leq n) \quad \text{(RecordModElim)}$$

$$\frac{\Gamma \vdash_K m : \sigma}{\Gamma \vdash_K (m : \sigma) : \sigma} \quad \text{(Coerce)}$$

$$\frac{\Gamma \vdash_K \lambda s{:}\sigma_1.ms : \Pi s{:}\sigma_1.\sigma_2}{\Gamma \vdash_K m : \Pi s{:}\sigma_1.\sigma_2} \ (s \notin m) \quad \text{(PiModEta)}$$

$$\frac{\Gamma \vdash_K \{\ell_i = \pi_{\ell_i}(m)^{[i=1\ldots n]}\} : \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\}}{\Gamma \vdash_K m : \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\}} \quad \text{(RecordModEta)}$$

Module Valuability

$$\frac{}{\Gamma \vdash_K s \downarrow \sigma} \ (\Gamma(s) = \sigma) \quad \text{(ModVarHalt)}$$

$$\frac{\Gamma \vdash_K c : \kappa}{\Gamma \vdash_K \langle c \rangle \downarrow \langle \kappa \rangle} \quad \text{(TypeModHalt)}$$

$$\frac{\Gamma \vdash_K e \downarrow c}{\Gamma \vdash_K \langle\!\langle e \rangle\!\rangle \downarrow \langle\!\langle c \rangle\!\rangle} \quad \text{(TermModHalt)}$$

$$\frac{\Gamma \vdash_K \sigma \ \text{sig} \quad \Gamma[s : \sigma_1] \vdash_K m : \sigma_2}{\Gamma \vdash_K \lambda s{:}\sigma_1.m \downarrow \Pi s{:}\sigma_1.\sigma_2} \quad \text{(PiModIntroHalt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_K m_i \downarrow \sigma_i[m_j^{[j=1\ldots i-1]}/s_j^{[j=1\ldots i-1]}] \\ \text{(for } 1 \leq i \leq n) \\ \Gamma \vdash_K \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\} \ \text{sig}\end{array}}{\Gamma \vdash_K \{\ell_i = m_i{}^{[i=1\ldots n]}\} \downarrow \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\}} \quad \text{(RecordModIntroHalt)}$$

$$\frac{\Gamma \vdash_K m \downarrow \{\ell_j \triangleright s_j : \sigma_j{}^{[i=1\ldots n]}\}}{\Gamma \vdash_K \pi_{\ell_i}(m) \downarrow \sigma_i[\pi_{\ell_j}(m)^{[j=1\ldots i-1]}/s_j^{[j=1\ldots i-1]}]} \quad (1 \leq i \leq n) \quad \text{(RecordModElimHalt)}$$

$$\frac{\Gamma \vdash_K m \downarrow \sigma}{\Gamma \vdash_K (m : \sigma) \downarrow \sigma} \quad \text{(CoerceHalt)}$$

Appendix B

Nuprl Proof Rules

Conventions

- Variables bound in a sequent are taken to be distinct. For example, Rule 2 has the implicit side condition that x not be bound by H, and Rule 3 has the implicit side-condition that x and y are distinct variables and neither is bound by H or J.
- Rules apply only when the consequent and subgoals are closed sequents. For example, Rule 2 has the implicit side-condition that x not appear free in H, and also (since, by the first convention, x cannot be bound by H) that x not appear free in J, C or t.
- When the inhabitant is omitted in the bottom (consequent) of a rule, the inhabitant is taken to be $\star$.
- Rules that may be derived from other rules are marked with a 'D'.

Structural Rules

$$\frac{}{H; x : T; J \vdash_\nu x \ \mathit{in}\ T} \tag{1}$$

$$\frac{H; J \vdash_\nu C \triangleleft t}{H; x : T; J \vdash_\nu C \triangleleft t} \tag{2}$$

$$\frac{H; y : T; x : S; J \vdash_\nu C \triangleleft t}{H; x : S; y : T; J \vdash_\nu C \triangleleft t} \tag{3}$$

$$\frac{H \vdash_\nu t \ \mathit{in}\ C}{H \vdash_\nu C \triangleleft t} \tag{4}$$

$$\frac{H \vdash_\nu C \triangleleft t}{H \vdash_\nu t \ \mathit{in}\ C} \tag{5}$$

$$\frac{H \vdash_\nu T \triangleleft t' \quad H; x : T \vdash_\nu C \triangleleft t}{H \vdash_\nu C[t'/x] \triangleleft t[t'/x]} \tag{D6}$$

Substitution

$$\frac{\begin{array}{c} H \vdash_\nu C[t'/x] \triangleleft s \\ H \vdash_\nu t = t' \ \mathit{in}\ T \qquad H; x : T \vdash_\nu C \ \text{type} \end{array}}{H \vdash_\nu C[t/x] \triangleleft s} \tag{7}$$

Direct Computation

$$\frac{H \vdash_\nu C[t'/x] \triangleleft s \quad \vdash_\nu t \sim t'}{H \vdash_\nu C[t/x] \triangleleft s} \tag{8}$$

$$\frac{H; x : T[t'/y]; J \vdash_\nu C \triangleleft s \quad \vdash_\nu t \sim t'}{H; x : T[t/y]; J \vdash_\nu C \triangleleft s} \tag{9}$$

$$\frac{}{\vdash_\nu t \sim t} \tag{10}$$

$$\frac{\vdash_\nu t' \sim t}{\vdash_\nu t \sim t'} \tag{11}$$

$$\frac{\vdash_\nu t_1 \sim t_2 \quad \vdash_\nu t_2 \sim t_3}{\vdash_\nu t_1 \sim t_3} \tag{12}$$

$$\frac{\vdash_\nu t_1 \sim t_1' \quad \vdash_\nu t_2 \sim t_2'}{\vdash_\nu t_1[t_2/x] \sim t_1'[t_2'/x]} \tag{13}$$

$$\frac{}{\vdash_\nu t \sim t'}\ (t \mapsto_s t') \tag{14}$$

The evaluation step relation is defined in Figure 4.3.

$$\frac{x \text{ appears actively in } t_2}{\vdash_\nu t_2[t_1/x] \sim \mathit{let}\ x = t_1 \ \mathit{in}\ t_2} \tag{15}$$

The active positions of a term are defined in Definition 4.5.

$$\frac{x_1 \notin t_2 \text{ and } x_2 \notin t_1}{\begin{array}{c}\vdash_\nu \mathit{let}\ x_1 = t_1 \ \mathit{in}\ \mathit{let}\ x_2 = t_2 \ \mathit{in}\ t \sim \\ \mathit{let}\ x_2 = t_2 \ \mathit{in}\ \mathit{let}\ x_1 = t_1 \ \mathit{in}\ t\end{array}} \tag{16}$$

Equality

$$\frac{H \vdash_\nu t_2 = t_1 \ \mathit{in}\ T}{H \vdash_\nu t_1 = t_2 \ \mathit{in}\ T} \tag{17}$$

$$\frac{H \vdash_\nu t_1 = t_2 \ \mathit{in}\ T \quad H \vdash_\nu t_2 = t_3 \ \mathit{in}\ T}{H \vdash_\nu t_1 = t_3 \ \mathit{in}\ T} \tag{18}$$

$$\frac{H \vdash_\nu t = t' \ \mathit{in}\ T}{H \vdash_\nu T \ \text{type}} \tag{19}$$

Subtyping

$$\frac{\begin{array}{c} H; x : T \vdash_\nu x \ \mathit{in}\ T' \\ H \vdash_\nu T \ \text{type} \qquad H \vdash_\nu T' \ \text{type} \end{array}}{H \vdash_\nu T \preceq T'} \tag{20}$$

$$\frac{H \vdash_\nu t_1 = t_2 \ \mathit{in}\ T' \quad H \vdash_\nu T' \preceq T}{H \vdash_\nu t_1 = t_2 \ \mathit{in}\ T} \tag{21}$$

$$\frac{H; x : T'; J \vdash_\nu C \triangleleft s \quad H \vdash_\nu T \preceq T'}{H; x : T; J \vdash_\nu C \triangleleft s} \tag{22}$$

$$\frac{H \vdash_\nu T \preceq T'}{H \vdash_\nu T \ \text{type}} \tag{23}$$

$$\frac{H \vdash_\nu T \preceq T'}{H \vdash_\nu T' \ \text{type}} \tag{24}$$

Universes

$$\frac{}{H \vdash_\nu \mathbb{U}_i \ \mathit{in}\ \mathbb{U}_j} \ (i < j) \tag{25}$$

$$\frac{H \vdash_\nu T \ \mathit{in}\ \mathbb{U}_i}{H \vdash_\nu T \ \mathit{in}\ \mathbb{U}_j} \ (i < j) \tag{26}$$

$$\frac{H \vdash_\nu T = T' \ \mathit{in}\ \mathbb{U}_i}{H \vdash_\nu T \preceq T'} \tag{27}$$

Void

$$\frac{}{H \vdash_\nu \mathit{Void} \ \mathit{in}\ \mathbb{U}_i} \tag{28}$$

$$\frac{H \vdash_\nu t \ \mathit{in}\ \mathit{Void}}{H \vdash_\nu C} \tag{29}$$

Atom

$$\frac{}{H \vdash_\nu \mathit{Atom} \ \mathit{in}\ \mathbb{U}_i} \tag{30}$$

$$\frac{}{H \vdash_\nu a \ \mathit{in}\ \mathit{Atom}} \ (\text{string literal } a) \tag{31}$$

$$\frac{H \vdash_\nu a_1 = a_1' \ \mathit{in}\ \mathit{Atom} \quad H \vdash_\nu a_2 = a_2' \ \mathit{in}\ \mathit{Atom}}{H \vdash_\nu (a_1 =_A a_2) = (a_1' =_A a_2') \ \mathit{in}\ \mathbb{B}} \tag{32}$$

$$\frac{H \vdash_\nu a_1 = a_2 \ \mathit{in}\ \mathit{Atom}}{H \vdash_\nu (a_1 =_A a_2) = \mathit{true} \ \mathit{in}\ \mathbb{B}} \tag{33}$$

$$\frac{H \vdash_\nu (a_1 =_A a_2) = \mathit{true} \ \mathit{in}\ \mathbb{B}}{H \vdash_\nu a_1 = a_2 \ \mathit{in}\ \mathit{Atom}} \tag{34}$$

Integer

$$\frac{}{H \vdash_\nu \mathbb{Z} \ \mathit{in}\ \mathbb{U}_i} \tag{35}$$

$$\frac{}{H \vdash_\nu n \ \mathit{in}\ \mathbb{Z}} \ (\text{integer literal } n) \tag{36}$$

$$\frac{\begin{array}{l} H \vdash_\nu C[0/x] \triangleleft b \\ H; x : \mathbb{Z}; y : (0 < x); z : C[x-1/x] \vdash_\nu C \triangleleft u \\ H; x : \mathbb{Z}; y : (x < 0); z : C[x+1/x] \vdash_\nu C \triangleleft d \end{array}}{\begin{array}{l} H; x : \mathbb{Z} \vdash_\nu C \triangleleft \\ \quad \mathit{fix}(\lambda f.\lambda x. \\ \qquad \mathit{if}\ x =_Z 0 \ \mathit{then}\ b \\ \qquad \mathit{else}\ \mathit{if}\ 0 \leq_Z x \ \mathit{then}\ u[\star, f(x-1)/y, z] \\ \qquad \mathit{else}\ d[\star, f(x+1)/y, z])\, x \end{array}} \tag{37}$$

$$\frac{H \vdash_\nu n_1 = n_1' \ \mathit{in}\ \mathbb{Z} \quad H \vdash_\nu n_2 = n_2' \ \mathit{in}\ \mathbb{Z}}{H \vdash_\nu (n_1 =_Z n_2) = (n_1' =_Z n_2') \ \mathit{in}\ \mathbb{B}} \tag{38}$$

$$\frac{H \vdash_\nu n_1 = n_2 \ \mathit{in}\ \mathbb{Z}}{H \vdash_\nu (n_1 =_Z n_2) = \mathit{true} \ \mathit{in}\ \mathbb{B}} \tag{39}$$

$$\frac{H \vdash_\nu (n_1 =_Z n_2) = \mathit{true} \ \mathit{in}\ \mathbb{B}}{H \vdash_\nu n_1 = n_2 \ \mathit{in}\ \mathbb{Z}} \tag{40}$$

$$\frac{H \vdash_\nu n_1 = n_1' \ \mathit{in}\ \mathbb{Z} \quad H \vdash_\nu n_2 = n_2' \ \mathit{in}\ \mathbb{Z}}{H \vdash_\nu (n_1 \leq_Z n_2) = (n_1' \leq_Z n_2') \ \mathit{in}\ \mathbb{B}} \tag{41}$$

$$\frac{H \vdash_\nu n_1 \leq n_2}{H \vdash_\nu (n_1 \leq_Z n_2) = \mathit{true} \ \mathit{in}\ \mathbb{B}} \tag{42}$$

$$\frac{H \vdash_\nu (n_1 \leq_Z n_2) = \mathit{true} \ \mathit{in}\ \mathbb{B}}{H \vdash_\nu n_1 \leq n_2} \tag{43}$$

$$\frac{H \vdash_\nu n_1 = n_1' \ in\ \mathbb{Z} \quad H \vdash_\nu n_2 = n_2' \ in\ \mathbb{Z}}{H \vdash_\nu n_1 \oplus n_2 = n_1' \oplus n_2' \ in\ \mathbb{Z}} \tag{44}$$

Not shown are the various rules governing the behavior of the binary operations $\oplus$.

Disjoint Union

$$\frac{H \vdash_\nu A = A' \ in\ \mathbb{U}_i \quad H \vdash_\nu B = B' \ in\ \mathbb{U}_i}{H \vdash_\nu A + B = A' + B' \ in\ \mathbb{U}_i} \tag{45}$$

$$\frac{H \vdash_\nu A \text{ type} \quad H \vdash_\nu B \text{ type}}{H \vdash_\nu A + B \text{ type}} \tag{46}$$

$$\frac{H \vdash_\nu a = a' \ in\ A \quad H \vdash_\nu A + B \text{ type}}{H \vdash_\nu inj_1(a) = inj_1(a') \ in\ A + B} \tag{47}$$

$$\frac{H \vdash_\nu b = b' \ in\ B \quad H \vdash_\nu A + B \text{ type}}{H \vdash_\nu inj_2(b) = inj_2(b') \ in\ A + B} \tag{48}$$

$$\frac{\begin{array}{l} H \vdash_\nu t = t' \ in\ A + B \\ H; x : A \vdash_\nu t_1 = t_1' \ in\ T \\ H; x : B \vdash_\nu t_2 = t_2' \ in\ T \end{array}}{H \vdash_\nu case(t, x.t_1, x.t_2) = case(t', x.t_1', x.t_2') \ in\ T} \tag{49}$$

$$\frac{\begin{array}{l} H; y : A; J[inj_1(y)/x] \\ \qquad \vdash_\nu C[inj_1(y)/x] \triangleleft s[inj_1(y)/x] \\ H; y : B; J[inj_2(y)/x] \\ \qquad \vdash_\nu C[inj_2(y)/x] \triangleleft s[inj_2(y)/x] \end{array}}{H; x : A + B; J \vdash_\nu C \triangleleft s} \tag{50}$$

Products

$$\frac{H \vdash_\nu A = A' \ in\ \mathbb{U}_i \quad H; x : A \vdash_\nu B = B' \ in\ \mathbb{U}_i}{H \vdash_\nu \Sigma x{:}A.B = \Sigma x{:}A'.B' \ in\ \mathbb{U}_i} \tag{51}$$

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu B \text{ type}}{H \vdash_\nu \Sigma x{:}A.B \text{ type}} \tag{52}$$

$$\frac{\begin{array}{c} H \vdash_\nu a = a' \ in\ A \\ H \vdash_\nu b = b' \ in\ B[a/x] \qquad H \vdash_\nu \Sigma x{:}A.B \text{ type} \end{array}}{H \vdash_\nu \langle a, b\rangle = \langle a', b'\rangle \ in\ \Sigma x{:}A.B} \tag{53}$$

$$\frac{H \vdash_\nu t = t' \ in\ \Sigma x{:}A.B}{H \vdash_\nu \pi_1(t) = \pi_1(t') \ in\ A} \tag{54}$$

$$\frac{H \vdash_\nu t = t' \ in\ \Sigma x{:}A.B}{H \vdash_\nu \pi_2(t) = \pi_2(t') \ in\ B[\pi_1(t)/x]} \tag{55}$$

$$\frac{\begin{array}{l} H; x : A; y : B; J[\langle x, y\rangle/z] \\ \qquad \vdash_\nu C[\langle x, y\rangle/z] \triangleleft s[\langle x, y\rangle/z] \end{array}}{H; z : \Sigma x{:}A.B; J \vdash_\nu C \triangleleft s} \tag{56}$$

$$\frac{\begin{array}{l} H \vdash_\nu \pi_1(p_1) = \pi_1(p_2) \ in\ A \\ H \vdash_\nu \pi_2(p_1) = \pi_2(p_2) \ in\ B[\pi_1(p_1)/x] \\ H \vdash_\nu p_1 \ in\ \Sigma x{:}A_1.B_1 \\ H \vdash_\nu p_2 \ in\ \Sigma x{:}A_2.B_2 \\ H \vdash_\nu \Sigma x{:}A.B \text{ type} \end{array}}{H \vdash_\nu p_1 = p_2 \ in\ \Sigma x{:}A.B} \tag{57}$$

Functions

$$\frac{H \vdash_\nu A = A' \ in\ \mathbb{U}_i \quad H; x : A \vdash_\nu B = B' \ in\ \mathbb{U}_i}{H \vdash_\nu \Pi x{:}A.B = \Pi x{:}A'.B' \ in\ \mathbb{U}_i} \tag{58}$$

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu B \text{ type}}{H \vdash_\nu \Pi x{:}A.B \text{ type}} \tag{59}$$

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu b = b' \ in\ B}{H \vdash_\nu \lambda x.b = \lambda x.b' \ in\ \Pi x{:}A.B} \tag{60}$$

$$\frac{H \vdash_\nu f = f' \ in\ \Pi x{:}A.B \quad H \vdash_\nu a = a' \ in\ A}{H \vdash_\nu f\, a = f'\, a' \ in\ B[a/x]} \tag{61}$$

$$\frac{\begin{array}{l} H; x : A \vdash_\nu f_1\, x = f_2\, x \ in\ B \\ H \vdash_\nu A \text{ type} \\ H \vdash_\nu f_1 \ in\ \Pi x{:}A_1.B_1 \\ H \vdash_\nu f_2 \ in\ \Pi x{:}A_2.B_2 \end{array}}{H \vdash_\nu f_1 = f_2 \ in\ \Pi x{:}A.B} \tag{62}$$

Very-Dependent Functions

$$\frac{\begin{array}{l} H \vdash_\nu A = A' \ in\ \mathbb{U}_i \\ H; y : A; z : A \vdash_\nu P \ in\ \mathbb{U}_i \\ H; x : A \vdash_\nu 0 \le e \\ H; y : A; z : A; w : P \vdash_\nu e[y/x] < e[z/x] \\ H; x : A; \\ \quad f : \{g \mid y : \{y : A \mid P[x/z]\} \to B[g, y/f, x]\} \\ \quad \vdash_\nu B = B' \ in\ \mathbb{U}_i \end{array}}{H \vdash_\nu \{f \mid x{:}A \to B\} = \{f \mid x{:}A' \to B'\} \ in\ \mathbb{U}_i} \tag{63}$$

$$\frac{\begin{array}{l} H \vdash_\nu A \;\text{ type} \\ H; y : A; z : A \vdash_\nu P \;\text{ type} \\ H; x : A \vdash_\nu 0 \leq e \\ H; y : A; z : A; w : P \vdash_\nu e[y/x] < e[z/x] \\ H; x : A; \\ \quad f : \{g \mid y : \{y : A \mid P[x/z]\} \to B[g, y/f, x]\} \\ \quad \vdash_\nu B \;\text{ type} \end{array}}{H \vdash_\nu \{f \mid x{:}A \to B\} \;\text{ type}} \tag{64}$$

$$\frac{\begin{array}{c} H \vdash_\nu \{f \mid x{:}A \to b\} \;\text{ type} \\ H; x : A \vdash_\nu b = b' \;\mathit{in}\; B[\lambda x.b/f] \end{array}}{H \vdash_\nu \lambda x.b = \lambda x.b' \;\mathit{in}\; \{f \mid x{:}A \to b\}} \tag{65}$$

$$\frac{H \vdash_\nu g = g' \;\mathit{in}\; \{f \mid x{:}A \to B\} \quad H \vdash_\nu a = a' \;\mathit{in}\; A}{H \vdash_\nu g\, a = g'\, a' \;\mathit{in}\; B[g, a/f, x]} \tag{66}$$

$$\frac{\begin{array}{l} H; x : A \vdash_\nu g_1 x = g_2 x \;\mathit{in}\; B[g_1/f] \\ H \vdash_\nu \{f \mid x{:}A \to B\} \;\text{ type} \\ H \vdash_\nu g_1 \;\mathit{in}\; \{f \mid x{:}A_1 \to B_1\} \\ H \vdash_\nu g_2 \;\mathit{in}\; \{f \mid x{:}A_2 \to B_2\} \end{array}}{H \vdash_\nu g_1 = g_2 \;\mathit{in}\; \{f \mid x{:}A \to B\}} \tag{67}$$

Partial Types

$$\frac{H \vdash_\nu T = T' \;\mathit{in}\; \mathbb{U}_i \quad H \vdash_\nu T \;\text{ total}}{H \vdash_\nu \overline{T} = \overline{T'} \;\mathit{in}\; \mathbb{U}_i} \tag{68}$$

$$\frac{H \vdash_\nu T \;\text{ total}}{H \vdash_\nu \overline{T} \;\text{ type}} \tag{69}$$

$$\frac{H \vdash_\nu t = t' \;\mathit{in}\; T \quad H \vdash_\nu \overline{T} \;\text{ type}}{H \vdash_\nu t = t' \;\mathit{in}\; \overline{T}} \tag{70}$$

$$\frac{\begin{array}{c} H; x : (t_1 \;\mathit{in!}\; T_1) \vdash_\nu t_1 = t_2 \;\mathit{in}\; T \\ H \vdash_\nu t_1 \;\mathit{in!}\; T_1 \Leftrightarrow t_2 \;\mathit{in!}\; T_2 \qquad H \vdash_\nu \overline{T} \;\text{ type} \end{array}}{H \vdash_\nu t_1 = t_2 \;\mathit{in}\; \overline{T}} \tag{71}$$

$$\frac{H \vdash_\nu f = f' \;\mathit{in}\; \overline{T} \to \overline{T} \quad H \vdash_\nu T \;\text{ admiss}}{H \vdash_\nu \mathit{fix}(f) = \mathit{fix}(f') \;\mathit{in}\; \overline{T}} \tag{72}$$

$$\frac{H \vdash_\nu T_1 \preceq T_2 \quad H \vdash_\nu T_2 \;\text{ total}}{H \vdash_\nu \overline{T_1} \preceq \overline{T_2}} \tag{D73}$$

Convergence

$$\frac{H \vdash_\nu t \;\mathit{in!}\; T' \quad H \vdash_\nu t \;\mathit{in}\; \overline{T}}{H \vdash_\nu t \;\mathit{in!}\; T} \tag{74}$$

$$\frac{H \vdash_\nu t \;\mathit{in}\; \overline{T}}{H \vdash_\nu t \;\mathit{in!}\; T} \;(t \text{ canonical}) \tag{75}$$

$$\frac{H \vdash_\nu t \;\mathit{in!}\; T}{H \vdash_\nu t \;\mathit{in}\; \overline{T}} \tag{76}$$

$$\frac{H \vdash_\nu t \;\mathit{in!}\; T' \quad H \vdash_\nu t = t' \;\mathit{in}\; \overline{T}}{H \vdash_\nu t = t' \;\mathit{in}\; T} \tag{77}$$

$$\frac{H; x : T \vdash_\nu x \;\mathit{in!}\; T' \quad H \vdash_\nu T \;\text{ type}}{H \vdash_\nu T \;\text{ total}} \tag{78}$$

$$\frac{H \vdash_\nu t \;\mathit{in}\; T \quad H \vdash_\nu T \;\text{ total}}{H \vdash_\nu t \;\mathit{in!}\; T} \tag{79}$$

$$\frac{H \vdash_\nu \mathit{let}\;\; x = t_1 \;\mathit{in}\; t_2 \;\mathit{in!}\; B \quad H \vdash_\nu t_1 \;\mathit{in}\; \overline{A}}{H \vdash_\nu t_1 \;\mathit{in!}\; A} \tag{80}$$

Sequencing

$$\frac{\begin{array}{c} H \vdash_\nu t_1 = t_1' \;\mathit{in}\; \overline{B} \\ H; x : B \vdash_\nu t_2 = t_2' \;\mathit{in}\; \overline{A} \qquad H \vdash_\nu \overline{A} \;\text{ type} \end{array}}{H \vdash_\nu (\mathit{let}\;\; x = t_1 \;\mathit{in}\; t_2) = (\mathit{let}\;\; x = t_1' \;\mathit{in}\; t_2') \;\mathit{in}\; \overline{A}} \tag{81}$$

$$\frac{H \vdash_\nu t_1 \;\mathit{in!}\; B \quad H; x : B \vdash_\nu t_2 \;\mathit{in}\; A}{H \vdash_\nu (\mathit{let}\;\; x = t_1 \;\mathit{in}\; t_2) = t_2[t_1/x] \;\mathit{in}\; A} \tag{82}$$

Set

$$\frac{\begin{array}{c} H \vdash_\nu A = A' \;\mathit{in}\; \mathbb{U}_i \\ H; x : A \vdash_\nu B \;\mathit{in}\; \mathbb{U}_i \\ H; x : A \vdash_\nu B' \;\mathit{in}\; \mathbb{U}_i \\ H; x : A \vdash_\nu B \Leftrightarrow B' \end{array}}{H \vdash_\nu \{x : A \mid B\} = \{x : A' \mid B'\} \;\mathit{in}\; \mathbb{U}_i} \tag{83}$$

$$\frac{H \vdash_\nu A \;\text{ type} \quad H; x : A \vdash_\nu B \;\text{ type}}{H \vdash_\nu \{x : A \mid B\} \;\text{ type}} \tag{84}$$

$$\frac{\begin{array}{c} H \vdash_\nu a = a' \;\mathit{in}\; A \\ H \vdash_\nu B[a/x] \qquad H \vdash_\nu \{x : A \mid B\} \;\text{ type} \end{array}}{H \vdash_\nu a = a' \;\mathit{in}\; \{x : A \mid B\}} \tag{85}$$

$$\frac{H;x:A \vdash_\nu B \ \text{type} \quad H;x:A;y:B;J \vdash_\nu C \triangleleft s}{H;x:\{x:A \mid B\};J \vdash_\nu C \triangleleft s} \tag{86}$$

Note that in Rule 86, y may not appear free in J, C or s, by convention.

Quotient

$$\frac{\begin{array}{l} H \vdash_\nu T = T' \ in\ \mathbb{U}_i \\ H;x:T;y:T \vdash_\nu E \ in\ \mathbb{U}_i \\ H;x:T;y:T \vdash_\nu E' \ in\ \mathbb{U}_i \\ H;x:T;y:T \vdash_\nu E \Leftrightarrow E' \\ H;x:T \vdash_\nu E[x/y] \\ H;x:T;y:T;z:E \vdash_\nu E[y,x/x,y] \\ H;x:T;y:T;z:T; \\ \quad w:E;w':E[y,z/x,y] \vdash_\nu E[z/y] \end{array}}{H \vdash_\nu xy{:}T//E = xy{:}T'//E' \ in\ \mathbb{U}_i} \tag{87}$$

$$\frac{\begin{array}{l} H \vdash_\nu T \ \text{type} \\ H;x:T;y:T \vdash_\nu E \ \text{type} \\ H;x:T \vdash_\nu E[x/y] \\ H;x:T;y:T;z:E \vdash_\nu E[y,x/x,y] \\ H;x:T;y:T;z:T; \\ \quad w:E;w':E[y,z/x,y] \vdash_\nu E[z/y] \end{array}}{H \vdash_\nu xy{:}T//E \ \text{type}} \tag{88}$$

$$\frac{H \vdash_\nu t = t' \ in\ T \quad H \vdash_\nu xy{:}T//E \ \text{type}}{H \vdash_\nu t = t' \ in\ xy{:}T//E} \tag{89}$$

$$\frac{\begin{array}{cc} H \vdash_\nu t \ in\ T & H \vdash_\nu t' \ in\ T \\ H \vdash_\nu e \ in\ E[t,t'/x,y] & H \vdash_\nu xy{:}T//E \ \text{type} \end{array}}{H \vdash_\nu t = t' \ in\ xy{:}T//E} \tag{90}$$

$$\frac{\begin{array}{l} H;x:A;y:A \vdash_\nu E \ \text{type} \\ H;x:(xy{:}T//E);J \vdash_\nu T \ \text{type} \\ H;x:A;y:A;z:B;J \vdash_\nu s = s'[y/x] \ in\ T \end{array}}{H;x:(xy{:}T//E);J \vdash_\nu s = s' \ in\ T} \tag{91}$$

Every Type

$$\frac{}{H \vdash_\nu \mathbb{E} \in \mathbb{U}_i} \tag{92}$$

$$\frac{H \vdash_\nu t \ in!\ T \quad H \vdash_\nu t' \ in!\ T'}{H \vdash_\nu t = t' \ in\ \mathbb{E}} \tag{93}$$

$$\frac{H \vdash_\nu t \ in\ T \quad H \vdash_\nu t' \ in\ T'}{H \vdash_\nu t = t' \ in\ Top} \tag{94}$$

$$\frac{H \vdash_\nu T \ \text{total}}{H \vdash_\nu \overline{T} \preceq \overline{\mathbb{E}}} \tag{D95}$$

Equality Type

$$\frac{\begin{array}{c} H \vdash_\nu T = T' \ in\ \mathbb{U}_i \\ H \vdash_\nu t_1 = t_1' \ in\ T \qquad H \vdash_\nu t_2 = t_2' \ in\ T \end{array}}{H \vdash_\nu (t_1 = t_2 \ in\ T) = (t_1' = t_2' \ in\ T') \ in\ \mathbb{U}_i} \tag{96}$$

$$\frac{H \vdash_\nu T \ \text{type} \quad H \vdash_\nu t_1 \ in\ T \quad H \vdash_\nu t_2 \ in\ T}{H \vdash_\nu (t_1 = t_2 \ in\ T) \ \text{type}} \tag{97}$$

$$\frac{H \vdash_\nu t = t' \ in\ T}{H \vdash_\nu \star \ in\ (t = t' \ in\ T)} \tag{98}$$

$$\frac{H;x:(t = t' \ in\ T);J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H;x:(t = t' \ in\ T);J \vdash_\nu C \triangleleft s} \tag{99}$$

Subtyping Type

$$\frac{H \vdash_\nu T_1 = T_1' \ in\ \mathbb{U}_i \quad H \vdash_\nu T_2 = T_2' \ in\ \mathbb{U}_i}{H \vdash_\nu (T_1 \preceq T_2) = (T_1' \preceq T_2') \ in\ \mathbb{U}_i} \tag{100}$$

$$\frac{H \vdash_\nu T_1 \ \text{type} \quad H \vdash_\nu T_2 \ \text{type}}{H \vdash_\nu (T_1 \preceq T_2) \ \text{type}} \tag{101}$$

$$\frac{H \vdash_\nu T \preceq T'}{H \vdash_\nu \star \ in\ (T \preceq T')} \tag{102}$$

$$\frac{H;x:(T \preceq T');J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H;x:(T \preceq T');J \vdash_\nu C \triangleleft s} \tag{103}$$

Inequality Type

$$\frac{H \vdash_\nu n_1 = n_1' \ in\ \mathbb{Z} \quad H \vdash_\nu n_2 = n_2' \ in\ \mathbb{Z}}{H \vdash_\nu (n_1 \le n_2) = (n_1' \le n_2') \ in\ \mathbb{U}_i} \tag{104}$$

$$\frac{H \vdash_\nu n \le n'}{H \vdash_\nu \star \ in\ (n \le n')} \tag{105}$$

$$\frac{H;x:(n \le n');J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H;x:(n \le n');J \vdash_\nu C \triangleleft s} \tag{106}$$

Convergence Type

$$\frac{H \vdash_\nu \overline{T} = \overline{T'} \ in \ \mathbb{U}_i \quad H \vdash_\nu t = t' \ in \ \overline{T}}{H \vdash_\nu (t \ in! \ T) = (t' \ in! \ T') \ in \ \mathbb{U}_i} \tag{107}$$

$$\frac{H \vdash_\nu t \ in \ \overline{T}}{H \vdash_\nu (t \ in! \ T) \ \text{type}} \tag{108}$$

$$\frac{H \vdash_\nu t \ in! \ T}{H \vdash_\nu \star \ in \ (t \ in! \ T)} \tag{109}$$

$$\frac{H; x : (t \ in! \ T); J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H; x : (t \ in! \ T); J \vdash_\nu C \triangleleft s} \tag{110}$$

Totality Type

$$\frac{H \vdash_\nu T = T' \ in \ \mathbb{U}_i}{H \vdash_\nu (T \ \text{total}) = (T' \ \text{total}) \ in \ \mathbb{U}_i} \tag{111}$$

$$\frac{H \vdash_\nu T \ \text{type}}{H \vdash_\nu (T \ \text{total}) \ \text{type}} \tag{112}$$

$$\frac{H \vdash_\nu T \ \text{total}}{H \vdash_\nu \star \ in \ (T \ \text{total})} \tag{113}$$

$$\frac{H; x : (T \ \text{total}); J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H; x : (T \ \text{total}); J \vdash_\nu C \triangleleft s} \tag{114}$$

Admissibility Type

$$\frac{H \vdash_\nu T = T' \ in \ \mathbb{U}_i}{H \vdash_\nu (T \ \text{admiss}) = (T' \ \text{admiss}) \ in \ \mathbb{U}_i} \tag{115}$$

$$\frac{H \vdash_\nu T \ \text{type}}{H \vdash_\nu (T \ \text{admiss}) \ \text{type}} \tag{116}$$

$$\frac{H \vdash_\nu T \ \text{admiss}}{H \vdash_\nu \star \ in \ (T \ \text{admiss})} \tag{117}$$

$$\frac{H; x : (T \ \text{admiss}); J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H; x : (T \ \text{admiss}); J \vdash_\nu C \triangleleft s} \tag{118}$$

Totality

$$\frac{H \vdash_\nu T \ \text{total}}{H \vdash_\nu T \ \text{type}} \tag{119}$$

$$\frac{}{H \vdash_\nu \mathbb{U}_i \ \text{total}} \tag{120}$$

$$\frac{}{H \vdash_\nu \mathit{Void} \ \text{total}} \tag{121}$$

$$\frac{}{H \vdash_\nu \mathit{Atom} \ \text{total}} \tag{122}$$

$$\frac{}{H \vdash_\nu \mathbb{Z} \ \text{total}} \tag{123}$$

$$\frac{H \vdash_\nu (A + B) \ \text{type}}{H \vdash_\nu (A + B) \ \text{total}} \tag{124}$$

$$\frac{H \vdash_\nu (\Sigma x{:}A.B) \ \text{type}}{H \vdash_\nu (\Sigma x{:}A.B) \ \text{total}} \tag{125}$$

$$\frac{H \vdash_\nu (\Pi x{:}A.B) \ \text{type}}{H \vdash_\nu (\Pi x{:}A.B) \ \text{total}} \tag{126}$$

$$\frac{H \vdash_\nu A \ \text{total} \quad H \vdash_\nu \{x : A \mid B\} \ \text{type}}{H \vdash_\nu \{x : A \mid B\} \ \text{total}} \tag{127}$$

$$\frac{H \vdash_\nu A \ \text{total} \quad H \vdash_\nu (xy{:}A/\!/B) \ \text{type}}{H \vdash_\nu (xy{:}A/\!/B) \ \text{total}} \tag{128}$$

$$\frac{}{H \vdash_\nu \mathbb{E} \ \text{total}} \tag{129}$$

$$\frac{H \vdash_\nu (t = t' \ in \ T) \ \text{type}}{H \vdash_\nu (t = t' \ in \ T) \ \text{total}} \tag{130}$$

$$\frac{H \vdash_\nu (T \preceq T') \ \text{type}}{H \vdash_\nu (T \preceq T') \ \text{total}} \tag{131}$$

$$\frac{H \vdash_\nu (t \ in! \ T) \ \text{type}}{H \vdash_\nu (t \ in! \ T) \ \text{total}} \tag{132}$$

$$\frac{H \vdash_\nu (T \ \text{total}) \ \text{type}}{H \vdash_\nu (T \ \text{total}) \ \text{total}} \tag{133}$$

$$\frac{H \vdash_\nu (T \ \text{admiss}) \ \text{type}}{H \vdash_\nu (T \ \text{admiss}) \ \text{total}} \tag{134}$$

Admissibility

$$\frac{H \vdash_\nu T \text{ admiss}}{H \vdash_\nu T \text{ type}} \tag{135}$$

$$\frac{}{H \vdash_\nu \mathit{Void} \text{ admiss}} \tag{136}$$

$$\frac{}{H \vdash_\nu \mathit{Atom} \text{ admiss}} \tag{137}$$

$$\frac{}{H \vdash_\nu \mathbb{Z} \text{ admiss}} \tag{138}$$

$$\frac{H \vdash_\nu A \text{ admiss} \quad H \vdash_\nu B \text{ admiss}}{H \vdash_\nu (A + B) \text{ admiss}} \tag{139}$$

$$\frac{H \vdash_\nu A \text{ admiss} \quad H \vdash_\nu B \text{ admiss}\,(x : A)}{H \vdash_\nu (\Sigma x{:}A.B) \text{ admiss}} \tag{140}$$

$$\frac{H; x : A \vdash_\nu B \text{ admiss} \quad H \vdash_\nu (\Pi x{:}A.B) \text{ type}}{H \vdash_\nu (\Pi x{:}A.B) \text{ admiss}} \tag{141}$$

$$\frac{H \vdash_\nu T \text{ admiss} \quad H \vdash_\nu \overline{T} \text{ type}}{H \vdash_\nu \overline{T} \text{ admiss}} \tag{142}$$

$$\frac{\begin{array}{l} H \vdash_\nu A \text{ admiss} \\ H \vdash_\nu B \text{ admiss}\,(x : A) \\ H; x : A; y : \downarrow B \vdash_\nu B \end{array}}{H \vdash_\nu \{x : A \mid B\} \text{ admiss}} \tag{143}$$

$$\frac{\begin{array}{l} H \vdash_\nu A \text{ admiss} \\ H \vdash_\nu B[\pi_1(z), \pi_2(z)/x, y] \text{ admiss}\,(z : A \times A) \\ H; x : A; y : A; z : \downarrow B \vdash_\nu B \\ H \vdash_\nu xy{:}A//B \text{ type} \end{array}}{H \vdash_\nu xy{:}A//B \text{ admiss}} \tag{144}$$

$$\frac{}{H \vdash_\nu \mathbb{E} \text{ admiss}} \tag{145}$$

$$\frac{H \vdash_\nu (t = t' \ \mathit{in}\ T) \text{ type}}{H \vdash_\nu (t = t' \ \mathit{in}\ T) \text{ admiss}} \tag{146}$$

$$\frac{H \vdash_\nu (T \preceq T') \text{ type}}{H \vdash_\nu (T \preceq T') \text{ admiss}} \tag{147}$$

$$\frac{H \vdash_\nu (t \leq t') \text{ type}}{H \vdash_\nu (t \leq t') \text{ admiss}} \tag{148}$$

$$\frac{H \vdash_\nu (t \ \mathit{in!}\ T) \text{ type}}{H \vdash_\nu (t \ \mathit{in!}\ T) \text{ admiss}} \tag{149}$$

$$\frac{H \vdash_\nu (T \text{ total}) \text{ type}}{H \vdash_\nu (T \text{ total}) \text{ admiss}} \tag{150}$$

$$\frac{H \vdash_\nu (T \text{ admiss}) \text{ type}}{H \vdash_\nu (T \text{ admiss}) \text{ admiss}} \tag{151}$$

More Admissibility Type

$$\frac{\begin{array}{l} H \vdash_\nu A = A' \ \mathit{in}\ \mathbb{U}_i \\ H; x : A \vdash_\nu B = B' \ \mathit{in}\ \mathbb{U}_i \end{array}}{\begin{array}{l} H \vdash_\nu (B \text{ admiss}\,(x : A)) = \\ \qquad (B' \text{ admiss}\,(x : A')) \ \mathit{in}\ \mathbb{U}_i \end{array}} \tag{152}$$

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu B \text{ type}}{H \vdash_\nu (B \text{ admiss}\,(x : A)) \text{ type}} \tag{153}$$

$$\frac{H \vdash_\nu B \text{ admiss}\,(x : A)}{H \vdash_\nu \star \ \mathit{in}\ (B \text{ admiss}\,(x : A))} \tag{154}$$

$$\frac{\begin{array}{l} H; x : (B \text{ admiss}\,(x : A)); J[\star/x] \\ \quad \vdash_\nu C[\star/x] \triangleleft s[\star/x] \end{array}}{H; x : (B \text{ admiss}\,(x : A)); J \vdash_\nu C \triangleleft s} \tag{155}$$

$$\frac{\begin{array}{l} H \vdash_\nu A = A' \ \mathit{in}\ \mathbb{U}_i \\ H; x : A \vdash_\nu B = B' \ \mathit{in}\ \mathbb{U}_i \end{array}}{\begin{array}{l} H \vdash_\nu (B \text{ wcoadmiss}\,(x : A)) = \\ \qquad (B' \text{ wcoadmiss}\,(x : A')) \ \mathit{in}\ \mathbb{U}_i \end{array}} \tag{156}$$

$$\frac{H \vdash_\nu A \text{ type} \quad H; x : A \vdash_\nu B \text{ type}}{H \vdash_\nu (B \text{ wcoadmiss}\,(x : A)) \text{ type}} \tag{157}$$

$$\frac{H \vdash_\nu B \text{ wcoadmiss}\,(x : A)}{H \vdash_\nu \star \ \mathit{in}\ (B \text{ wcoadmiss}\,(x : A))} \tag{158}$$

$$\frac{\begin{array}{l} H; x : (B \text{ wcoadmiss}\,(x : A)); J[\star/x] \\ \quad \vdash_\nu C[\star/x] \triangleleft s[\star/x] \end{array}}{H; x : (B \text{ wcoadmiss}\,(x : A)); J \vdash_\nu C \triangleleft s} \tag{159}$$

$$\frac{\begin{array}{c}H \vdash_\nu A = A' \ in\ \mathbb{U}_i \\ H; x : A \vdash_\nu B = B' \ in\ \mathbb{U}_i\end{array}}{\begin{array}{l}H \vdash_\nu (B \operatorname{coadmiss} (x : A)) = \\ \qquad (B' \operatorname{coadmiss} (x : A'))\ in\ \mathbb{U}_i\end{array}} \quad (160)$$

$$\frac{H \vdash_\nu A \ \text{type} \quad H; x : A \vdash_\nu B \ \text{type}}{H \vdash_\nu (B \operatorname{coadmiss} (x : A)) \ \text{type}} \quad (161)$$

$$\frac{H \vdash_\nu B \operatorname{coadmiss} (x : A)}{H \vdash_\nu \star\ in\ (B \operatorname{coadmiss} (x : A))} \quad (162)$$

$$\frac{\begin{array}{c}H; x : (B \operatorname{coadmiss} (x : A)); J[\star/x] \\ \vdash_\nu C[\star/x] \triangleleft s[\star/x]\end{array}}{H; x : (B \operatorname{coadmiss} (x : A)); J \vdash_\nu C \triangleleft s} \quad (163)$$

$$\frac{H \vdash_\nu T = T' \ in\ \mathbb{U}_i}{H \vdash_\nu (T \ \text{mono}) = (T' \ \text{mono})\ in\ \mathbb{U}_i} \quad (164)$$

$$\frac{H \vdash_\nu T \ \text{type}}{H \vdash_\nu (T \ \text{mono}) \ \text{type}} \quad (165)$$

$$\frac{H \vdash_\nu T \ \text{mono}}{H \vdash_\nu \star\ in\ (T \ \text{mono})} \quad (166)$$

$$\frac{H; x : (T \ \text{mono}); J[\star/x] \vdash_\nu C[\star/x] \triangleleft s[\star/x]}{H; x : (T \ \text{mono}); J \vdash_\nu C \triangleleft s} \quad (167)$$

More Admissibility

$$\frac{H \vdash_\nu A \operatorname{admiss} (x : T) \quad H \vdash_\nu T}{H \vdash_\nu A \ \text{admiss}} \ (x \notin A) \quad (168)$$

$$\frac{H \vdash_\nu A \ \text{admiss} \quad H \vdash_\nu T \ \text{type}}{H \vdash_\nu A \operatorname{admiss} (x : T)} \ (x \notin A) \quad (169)$$

$$\frac{H \vdash_\nu A \ \text{type} \quad H \vdash_\nu T \ \text{type}}{H \vdash_\nu A \operatorname{wcoadmiss} (x : T)} \ (x \notin A) \quad (170)$$

$$\frac{H \vdash_\nu A \operatorname{coadmiss} (x : T)}{H \vdash_\nu A \operatorname{wcoadmiss} (x : T)} \quad (171)$$

$$\frac{H \vdash_\nu A \ \text{mono}}{H \vdash_\nu A \ \text{admiss}} \quad (172)$$

$$\frac{H \vdash_\nu T \ \text{type}}{H \vdash_\nu \mathit{Void} \operatorname{coadmiss} (x : T)} \quad (173)$$

$$\frac{}{H \vdash_\nu \mathit{Void} \ \text{mono}} \quad (174)$$

$$\frac{H \vdash_\nu T \ \text{type}}{H \vdash_\nu \mathit{Atom} \operatorname{coadmiss} (x : T)} \quad (175)$$

$$\frac{}{H \vdash_\nu \mathit{Atom} \ \text{mono}} \quad (176)$$

$$\frac{H \vdash_\nu T \ \text{type}}{H \vdash_\nu \mathbb{Z} \operatorname{coadmiss} (x : T)} \quad (177)$$

$$\frac{}{H \vdash_\nu \mathbb{Z} \ \text{mono}} \quad (178)$$

$$\frac{\begin{array}{c}H \vdash_\nu A \operatorname{admiss} (x : T) \\ H \vdash_\nu B \operatorname{admiss} (x : T)\end{array}}{H \vdash_\nu (A + B) \operatorname{admiss} (x : T)} \quad (179)$$

$$\frac{\begin{array}{c}H \vdash_\nu A \operatorname{wcoadmiss} (x : T) \\ H \vdash_\nu B \operatorname{wcoadmiss} (x : T)\end{array}}{H \vdash_\nu (A + B) \operatorname{wcoadmiss} (x : T)} \quad (180)$$

$$\frac{\begin{array}{c}H \vdash_\nu A \operatorname{coadmiss} (x : T) \\ H \vdash_\nu B \operatorname{coadmiss} (x : T)\end{array}}{H \vdash_\nu (A + B) \operatorname{coadmiss} (x : T)} \quad (181)$$

$$\frac{H \vdash_\nu A \ \text{mono} \quad H \vdash_\nu B \ \text{mono}}{H \vdash_\nu (A + B) \ \text{mono}} \quad (182)$$

$$\frac{\begin{array}{l}H \vdash_\nu A \operatorname{admiss} (y : T) \\ H \vdash_\nu B[\pi_1(z), \pi_2(z)/y, x] \operatorname{admiss} (z : \Sigma y{:}T.A)\end{array}}{H \vdash_\nu (\Sigma x{:}A.B) \operatorname{admiss} (y : T)} \quad (183)$$

$$\frac{\begin{array}{l}H \vdash_\nu A \operatorname{wcoadmiss} (y : T) \\ H; y' : T; x : A[y'/y] \vdash_\nu B \operatorname{wcoadmiss} (y : T) \\ H; y : T \vdash_\nu \Sigma x{:}A.B \ \text{type}\end{array}}{H \vdash_\nu (\Sigma x{:}A.B) \operatorname{wcoadmiss} (y : T)} \quad (184)$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ coadmiss } (y:T)\\ H \vdash_\nu B[\pi_1(z),\pi_2(z)/y,x] \text{ coadmiss } (z:\Sigma y{:}T.A)\end{array}}{H \vdash_\nu (\Sigma x{:}A.B) \text{ coadmiss } (y:T)} \tag{185}$$

$$\frac{H \vdash_\nu A \ \text{ mono} \quad H;x:A \vdash_\nu B \ \text{ mono}}{H \vdash_\nu (\Sigma x{:}A.B) \ \text{ mono}} \tag{186}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ wcoadmiss } (y:T)\\ H;y':T;x:A[y'/y] \vdash_\nu B \text{ admiss } (y:T)\\ H;y:T \vdash_\nu \Pi x{:}A.B \ \text{ type}\end{array}}{H \vdash_\nu (\Pi x{:}A.B) \text{ admiss } (y:T)} \tag{187}$$

$$\frac{H \vdash_\nu A \ \text{ mono} \quad H;x:A \vdash_\nu B \ \text{ mono}}{H \vdash_\nu (\Pi x{:}A.B) \ \text{ mono}} \tag{188}$$

$$\frac{H \vdash_\nu A \text{ admiss } (x:T) \quad H;x:T \vdash_\nu \overline{A} \ \text{ type}}{H \vdash_\nu \overline{A} \text{ admiss } (x:T)} \tag{189}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ admiss } y:T\\ H \vdash_\nu B[\pi_1(z),\pi_2(z)/y,x] \text{ admiss } (z:\Sigma y{:}T.A)\\ H;y:T;x:A;z:\downarrow B \vdash_\nu B\\ H;y:T \vdash_\nu \{x:A \mid B\} \ \text{ type}\end{array}}{H \vdash_\nu \{x:A \mid B\} \text{ admiss } (y:T)} \tag{190}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ wcoadmiss } (y:T)\\ H;y':T;x:A[y'/y] \vdash_\nu B \text{ wcoadmiss } (y:T)\\ H;y:T \vdash_\nu \{x{:}A \mid B\} \ \text{ type}\end{array}}{H \vdash_\nu \{x:A \mid B\} \text{ wcoadmiss } (y:T)} \tag{191}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ coadmiss } (y:T)\\ H \vdash_\nu B[\pi_1(z),\pi_2(z)/y,x] \text{ coadmiss } (z:\Sigma y{:}T.A)\end{array}}{H \vdash_\nu \{x:A \mid B\} \text{ coadmiss } (y:T)} \tag{192}$$

$$\frac{H \vdash_\nu A \ \text{ mono} \quad H \vdash_\nu \{x:A \mid B\} \ \text{ type}}{H \vdash_\nu \{x:A \mid B\} \ \text{ mono}} \tag{193}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ admiss } z:T\\ H \vdash_\nu B[\pi_1(w),\pi_1(\pi_2(w)),\pi_2(\pi_2(w))/z,x,y]\\ \quad \text{admiss}(w:\Sigma z{:}T.(A\times A))\\ H;z:T;x:A;y:A;w:\downarrow B \vdash_\nu B\\ H;y:T \vdash_\nu xy{:}A//B \ \text{ type}\end{array}}{H \vdash_\nu xy{:}A//B \text{ admiss } (z:T)} \tag{194}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ wcoadmiss } (z:T)\\ H;z':T;x:A[z'/z];y:A[z'/z] \vdash_\nu\\ \quad B \text{ wcoadmiss } (z:T)\\ H;z:T \vdash_\nu xy{:}A//B \ \text{ type}\end{array}}{H \vdash_\nu xy{:}A//B \text{ wcoadmiss } (z:T)} \tag{195}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ coadmiss } (z:T)\\ H \vdash_\nu B[\pi_1(w),\pi_1(\pi_2(w)),\pi_2(\pi_2(w))/z,x,y]\\ \quad \text{coadmiss}(w:\Sigma y{:}T.(A\times A))\end{array}}{H \vdash_\nu xy{:}A//B \text{ coadmiss } (z:T)} \tag{196}$$

$$\frac{H \vdash_\nu A \ \text{ mono} \quad H \vdash_\nu xy{:}A//B \ \text{ type}}{H \vdash_\nu xy{:}A//B \ \text{ mono}} \tag{197}$$

$$\frac{H \vdash_\nu T \ \text{ type}}{H \vdash_\nu \mathbb{E} \text{ coadmiss } (x:T)} \tag{198}$$

$$\frac{}{H \vdash_\nu \mathbb{E} \ \text{ mono}} \tag{199}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ admiss } (x:T)\\ H;x:T \vdash_\nu (a=a' \ in \ A) \ \text{ type}\end{array}}{H \vdash_\nu (a=a' \ in \ A) \text{ admiss } (x:T)} \tag{200}$$

$$\frac{\begin{array}{l}H \vdash_\nu A \text{ coadmiss } (x:T)\\ H;x:T \vdash_\nu (a=a' \ in \ A) \ \text{ type}\end{array}}{H \vdash_\nu (a=a' \ in \ A) \text{ coadmiss } (x:T)} \tag{201}$$

$$\frac{H \vdash_\nu (a=a' \ in \ A) \ \text{ type}}{H \vdash_\nu (a=a' \ in \ A) \ \text{ mono}} \tag{202}$$

$$\frac{H;x:T \vdash_\nu (a \le a') \ \text{ type} \quad H \vdash_\nu T \ \text{ type}}{H \vdash_\nu (a \le a') \text{ admiss } (x:T)} \tag{203}$$

Classical Rules

$$\frac{H; x:T \vdash_\nu (a \leq a')\ \text{type} \quad H \vdash_\nu T\ \text{type}}{H \vdash_\nu (a \leq a')\ \text{coadmiss}\ (x:T)} \tag{204}$$

$$\frac{H \vdash_\nu (a \leq a')\ \text{type}}{H \vdash_\nu (a \leq a'\ \mathit{in}\ A)\ \text{mono}} \tag{205}$$

$$\frac{H; x:T \vdash_\nu (a\ \mathit{in!}\ A)\ \text{type} \quad H \vdash_\nu T\ \text{type}}{H \vdash_\nu (a\ \mathit{in!}\ A)\ \text{admiss}\ (x:T)} \tag{206}$$

$$\frac{H; x:T \vdash_\nu (a\ \mathit{in!}\ A)\ \text{type} \quad H \vdash_\nu T\ \text{type}}{H \vdash_\nu (a\ \mathit{in!}\ A)\ \text{coadmiss}\ (x:T)} \tag{207}$$

$$\frac{H \vdash_\nu (a\ \mathit{in!}\ A)\ \text{type}}{H \vdash_\nu (a\ \mathit{in!}\ A)\ \text{mono}} \tag{208}$$

$$\frac{\begin{array}{l} H; y:T \vdash_\nu d\ \mathit{in}\ T_1 + T_2 \\ H; y:T; x:T_1 \vdash_\nu A\ \text{type} \\ H; y:T; x:T_2 \vdash_\nu B\ \text{type} \\ H \vdash_\nu A[\pi_1(z), \pi_2(z)/y, x] \\ \qquad\qquad \text{admiss}(z : \Sigma y{:}T.T_1) \\ H \vdash_\nu B[\pi_1(z), \pi_2(z)/y, x] \\ \qquad\qquad \text{admiss}(z : \Sigma y{:}T.T_2) \end{array}}{H \vdash_\nu \mathit{case}(d, x.A, x.B)\ \text{admiss}\ (y:T)} \tag{209}$$

$$\frac{\begin{array}{l} H; y:T \vdash_\nu d\ \mathit{in}\ T_1 + T_2 \\ H; y:T; x:T_1 \vdash_\nu A\ \text{type} \\ H; y:T; x:T_2 \vdash_\nu B\ \text{type} \\ H \vdash_\nu A[\pi_1(z), \pi_2(z)/y, x] \\ \qquad\qquad \text{wcoadmiss}(z : \Sigma y{:}T.T_1) \\ H \vdash_\nu B[\pi_1(z), \pi_2(z)/y, x] \\ \qquad\qquad \text{wcoadmiss}(z : \Sigma y{:}T.T_2) \end{array}}{H \vdash_\nu \mathit{case}(d, x.A, x.B)\ \text{wcoadmiss}\ (y:T)} \tag{210}$$

$$\frac{\begin{array}{l} H; y:T \vdash_\nu d\ \mathit{in}\ T_1 + T_2 \\ H; y:T; x:T_1 \vdash_\nu A\ \text{type} \\ H; y:T; x:T_2 \vdash_\nu B\ \text{type} \\ H \vdash_\nu A[\pi_1(z), \pi_2(z)/y, x] \\ \qquad\qquad \text{coadmiss}(z : \Sigma y{:}T.T_1) \\ H \vdash_\nu B[\pi_1(z), \pi_2(z)/y, x] \\ \qquad\qquad \text{coadmiss}(z : \Sigma y{:}T.T_2) \end{array}}{H \vdash_\nu \mathit{case}(d, x.A, x.B)\ \text{coadmiss}\ (y:T)} \tag{211}$$

$$\frac{}{H \vdash \forall P{:}\mathbb{P}_i.\downarrow(P \vee \neg P) \triangleleft \lambda x.\star} \tag{XM}$$

$$\frac{H \vdash_\nu A\ \text{wcoadmiss}\ (x:T) \quad H; x:T \vdash_\nu \overline{A}\ \text{type}}{H \vdash_\nu \overline{A}\ \text{wcoadmiss}\ (x:T)} \tag{PWC}$$

$$\frac{H \vdash_\nu A\ \text{coadmiss}\ (x:T) \quad H; x:T \vdash_\nu \overline{A}\ \text{type}}{H \vdash_\nu \overline{A}\ \text{coadmiss}\ (x:T)} \tag{PC}$$

$$\frac{\begin{array}{l} H \vdash_\nu A = A'\ \mathit{in}\ \mathbb{U}_i \\ H; y:A; z:A \vdash_\nu P\ \mathit{in}\ \mathbb{U}_i \\ H \vdash_\nu \neg(\exists f{:}\mathbb{N} \to A.\, \forall n{:}\mathbb{N}.\, P[f(n+1), f\,n/y, z]) \\ H; x:A; \\ \quad f : \{g \mid y : \{y : A \mid P[x/z]\} \to B[g, y/f, x]\} \\ \quad \vdash_\nu B = B'\ \mathit{in}\ \mathbb{U}_i \end{array}}{H \vdash_\nu \{f \mid x{:}A \to B\} = \{f \mid x{:}A' \to B'\}\ \mathit{in}\ \mathbb{U}_i} \tag{VF}$$

$$\frac{\begin{array}{l} H \vdash_\nu A\ \text{type} \\ H; y:A; z:A \vdash_\nu P\ \text{type} \\ H \vdash_\nu \neg(\exists f{:}\mathbb{N} \to A.\, \forall n{:}\mathbb{N}.\, P[f(n+1), f\,n/y, z]) \\ H; x:A; \\ \quad f : \{g \mid y : \{y : A \mid P[x/z]\} \to B[g, y/f, x]\} \\ \quad \vdash_\nu B\ \text{type} \end{array}}{H \vdash_\nu \{f \mid x{:}A \to B\}\ \text{type}} \tag{VFT}$$

Appendix C

Proofs

Lemma C.1 *If $T \in \mathbb{U}_i$ and $T \sim T'$ then $T = T' \in \mathbb{U}_i$.*

Proof

It is easy to verify that the statement holds for all T if it holds for canonical T. Therefore I assume, without loss of generality, that T is canonical and prove the statement by induction on the structure of T. I show the function and partial type cases, the other cases are similar.

Suppose $T \in \mathbb{U}_i$ and $T \sim T'$ where $T \equiv \Pi x{:}A.B$. Then (for some A', B') $T' \Downarrow \Pi x{:}A'.B'$ where $A \sim A'$ and $B \sim B'$. By induction $A = A' \in \mathbb{U}_i$. Suppose $a = a' \in A$. Then $B[a/x] = B[a'/x] \in \mathbb{U}_i$ and $B[a'/x] \sim B'[a'/x]$ so, by induction, $B[a'/x] = B'[a'/x] \in \mathbb{U}_i$. It follows that $B[a/x] = B'[a'/x] \in \mathbb{U}_i$ so $T = T' \in \mathbb{U}_i$.

Suppose $T \in \mathbb{U}_i$ and $T \sim T'$ where $T \equiv \overline{A}$. Then (for some A') $T' \Downarrow \overline{A'}$ where $A \sim A'$. Since $A \in \mathbb{U}_i$, it follows by induction that $A = A' \in \mathbb{U}_i$. Hence $\overline{A} = \overline{A'} \in \mathbb{U}_i$ so $T = T' \in \mathbb{U}_i$. □

Theorem 4.3 If T type and $T \sim T'$ then $T = T'$. If $t \in T$ and $t \sim t'$ then $t = t' \in T$.

Proof

The first part is shown by a similar argument to Lemma C.1. For the second part, it is easy to verify that the statement holds for all T if it holds for canonical T. Therefore I assume, without loss of generality, that T is canonical and prove the statement by induction on the structure of T. The case where $T \equiv \mathbb{U}_i$ follows from Lemma C.1. I show the function and partial type cases, the other cases are similar.

Suppose $t \in \Pi x{:}A.B$ and $t \sim t'$. Then (for some b) $t \Downarrow \lambda x.b$. Therefore (for some b') $t' \Downarrow \lambda x.b'$ and $b \sim b'$. Suppose $a = a' \in A$. Then $b[a/x] = b[a'/x] \in B[a/x]$ and $b[a'/x] \sim b'[a'/x]$ so, by induction, $b[a'/x] = b'[a'/x] \in B[a/x]$. It follows that $b[a/x] = b'[a'/x] \in B[a/x]$ so $t = t' \in \Pi x{:}A.B$.

Suppose $t \in \overline{T}$ and $t \sim t'$. Certainly $t\downarrow \Leftrightarrow t'\downarrow$. Suppose $t\downarrow$, then $t \in T$. By induction $t = t' \in T$ and hence $t = t' \in \overline{T}$. □

Lemma C.2 *If x appears actively in t and $t[e/x]\downarrow$ then $e\downarrow$.*

Proof

By induction on the structure of t.

Lemma 4.6 If x appears actively in t then $t[e/x] \sim let\ x = e\ in\ t$.

Proof

This proof makes use of Proposition 5.4 and Lemma 5.5, but those facts are shown without using this lemma. I first show that $let\ x = e\ in\ t \leq t[e/x]$. Let σ be a substitution such that $\sigma(let\ x = e\ in\ t)$ and $\sigma(t[e/x])$ are closed and suppose, without loss of generality, that σ does not substitute for x. Suppose further that $\sigma(let\ x = e\ in\ t)\downarrow$. Then $\sigma(e)\downarrow$ so let $\sigma(e) \Downarrow v$. Note that $\sigma(e) \sim v$. Therefore $\sigma(let\ x = e\ in\ t) \equiv let\ x = \sigma(e)\ in\ \sigma(t) \leq let\ x = v\ in\ \sigma(t) \leq \sigma(t)[v/x] \leq \sigma(t)[\sigma(e)/x] \equiv \sigma(t[e/x])$. Hence $let\ x = e\ in\ t \leq e[t/x]$.

I now show that $t[e/x] \leq let\ x = e\ in\ t$. Again suppose $\sigma(let\ x = e\ in\ t)$ and $\sigma(t[e/x])$ are closed and suppose, without loss of generality, that σ does not substitute for x. Suppose further that $\sigma(t[e/x])\downarrow$. Note that the active positions of t are also active in $\sigma(t)$. By Lemma C.2, $\sigma(e)\downarrow$. Thus $\sigma(t[e/x]) \leq \sigma(let\ x = e\ in\ t)$ in a similar manner as before. $\square$

Lemma 4.10 If τ is a type system, then for each $\text{OP} \in \text{TYPES}$, $\text{OP}(\tau)$ is a type system.

Proof

By tedious verification of all cases. The only subtle argument is for unique valuation: For each operator other than VFUN, the existential prefix's instances are uniquely given by τ, T and T'. (Parametric variables (*e.g.*, $\beta_{(-)}$) are unique over the membership of α.) Therefore ϕ is uniquely given for operators other than VFUN.

For the VFUN operator the term P is not uniquely given by τ, T and T'. The relations $<$, $\rho_{(-,-)}$, and $\gamma_{(-)}$ also are not, but they are unique when also given P. Further, it is not obvious that $\delta_{(-,-)}$ is uniquely given by τ, T and T', but again it is clearly unique when also given P. Suppose ϕ and ϕ' are different equalities resulting from terms P and P'. Define $<$ and $<'$, γ and γ', and so forth, to be the various relations resulting from P and P'. We wish to show that $t\ \phi\ t'$ implies $t\ \phi'\ t'$. Suppose $t\ \phi\ t'$. Then $t \Downarrow \lambda x.b$ and $t' \Downarrow \lambda x.b'$. It remains to show that $b[a/x]\ \delta_{(t,a)}\ b'[a'/x]$ implies $b[a/x]\ \delta'_{(t,a)}\ b'[a'/x]$ whenever $a\ \alpha\ a'$. However, $\delta_{(t,a)}$ and $\delta'_{(t,a)}$ are identical relations when they are both defined, so it suffices to show that $\delta'_{(t,a)}$ is defined for all $a\ \alpha\ a$. This may be shown by induction using the order $<'$. $\square$

Lemma 4.11 If

- τ is type symmetric and type transitive, and
- τ defines only universes (that is, if $\tau T T \phi$ then $T \Downarrow \mathbb{U}_i$ for some i), and
- $\text{CLOSE}(\tau) T_1 T'_1 \phi_1$ and $\text{CLOSE}(\tau) T_2 T'_2 \phi_2$, and
- either $T_1 \equiv T_2$ or $T_1 \equiv T'_2$,

then either $\tau T_1 T'_1 \phi_1$ and $\tau T_2 T'_2 \phi_2$, or for some $\text{OP} \in \text{TYPES}$, $\text{OP}(\text{CLOSE}(\tau)) T_1 T'_1 \phi_1$ and $\text{OP}(\text{CLOSE}(\tau)) T_2 T'_2 \phi_2$.

Proof

By inspection.

Lemma 4.12 If τ is a type system and if τ defines only universes, then $\textsc{Close}(\tau)$ is a type system.

Proof

All but unique valuation and type transitivity are directly by Lemma 4.10 and induction. Unique valuation and type transitivity are by Lemma 4.11 (to ensure both antecedents are from the same case), then by Lemma 4.10 and induction. □

Lemma 4.13 For all i, $\textsc{Univ}_i$ and $\textsc{Nuprl}_i$ are type systems.

Proof

By complete induction on i.

Theorem 5.1 There exist inadmissible types.

Proof

This example is due to Smith [92]. Let the type T and the function f be defined as follows:

$$\begin{array}{rcl} T & \stackrel{\text{def}}{=} & \Sigma h{:}(\mathbb{N} \to \overline{\mathbb{N}}).\, ((\Pi x{:}\mathbb{N}.\, h\, x\ \mathit{in}!\ \mathbb{N}) \to \mathit{Void}) \\ f & \stackrel{\text{def}}{=} & \lambda p.\langle g, \lambda y.\star\rangle \\ g & \stackrel{\text{def}}{=} & \lambda x.\, \mathit{if}\ x \leq 0\ \mathit{then}\ 0\ \mathit{else}\ \pi_1(p)(x-1) \end{array}$$

It is easy to verify that $\overline{T}$ type. We wish to show that f has type $\overline{T} \to \overline{T}$. Suppose $t = t' \in \overline{T}$. We need $g[t/p] = g[t'/p] \in \mathbb{N} \to \overline{\mathbb{N}}$ and $\lambda y.\star \in (\Pi x{:}\mathbb{N}.\,(g[t/p])x\ \mathit{in}!\ \mathbb{N}) \to \mathit{Void}$. The former is easily shown; to show the latter, I assume that $(g[t/p])n$ converges for every natural number n and draw a contradiction. It follows that $\Pi x{:}\mathbb{N}.\,(g[t/p])x\ \mathit{in}!\ \mathbb{N}$ is empty and $\lambda y.\star$ is vacuously a function from any empty type to Void. Suppose $(g[t/p])n$ converges for every natural number n. Then the term $\mathit{if}\ n \leq 0\ \mathit{then}\ 0\ \mathit{else}\ \pi_1(t)(n-1)$ also converges for every natural number n. It follows that $t\!\downarrow$, since $\pi_1(t)(0)\!\downarrow$, and hence $t \in T$. Thus $(\Pi x{:}\mathbb{N}.\,\pi_1(t)(x)\ \mathit{in}!\ \mathbb{N}) \to \mathit{Void}$ is inhabited (by $\pi_2(t)$) and consequently it cannot be the case that $\pi_1(t)(n)\!\downarrow$ for every natural number n. But this is a contradiction since $\pi_1(t)(n-1)\!\downarrow$ for every $n > 1$. Therefore $f \in \overline{T} \to \overline{T}$.

However, it is not the case that $\mathit{fix}(f) \in \overline{T}$. Suppose $\mathit{fix}(f) \in \overline{T}$. Then $\mathit{fix}(f) \in T$ since $\mathit{fix}(f)$ converges (in two steps). Thus $\pi_2(\mathit{fix}(f)) \in (\Pi x{:}\mathbb{N}.\,\pi_1(\mathit{fix}(f))(x)\ \mathit{in}!\ \mathbb{N}) \to \mathit{Void}$, which implies that $\pi_1(\mathit{fix}(f))$ is not total on $\mathbb{N}$, but it is easy to show by induction that $\pi_1(\mathit{fix}(f))$ is in fact total (on $\mathbb{N}$). Therefore $\mathit{fix}(f) \notin \overline{T}$ and hence T is not admissible. □

Lemma 5.5 If $e \leq e'$ and $t \leq t'$ then $e[t/x] \leq e'[t'/x]$.

Proof

The proof is a straightforward application of Howe's method. In order to keep this proof brief, I draw from the terminology and results of Howe's paper [53] and will not repeat the definitions or lemmas. I write $t \Downarrow^k t'$ when $t \Downarrow t'$ in k steps and $t \Downarrow^{<k} t'$ when $t \Downarrow t'$ in less than k steps. It is sufficient to show that each of the noncanonical operators are extensional.

Case 1: Suppose $t \equiv e_1 e_2$, $t' \equiv e_1' e_2'$ and a are closed. Suppose further that $t \Downarrow^k a$, $e_i \leq^* e_i'$ and the induction hypothesis holds (for every closed u_1, u_2 and u, if $u_1 \Downarrow^{<k} u_2$ and $u_1 \leq^* u$ then $u_2 \leq^* u$). Then $e_1 \Downarrow^{<k} \lambda x.b$ and $b[e_2/x] \Downarrow^{<k} a$. By the induction hypothesis, $\lambda x.b \leq^* e_1'$. By Howe's Lemma 2, there exists $b' \geq^* b$ such that $e_1' \Downarrow \lambda x.b'$. By

Howe's Lemma 1, $b[e_2/x] \leq^* b'[e_2'/x]$. By the induction hypothesis $a \leq^* b'[e_2'/x]$. However, $t' \mapsto^* b'[e_2'/x]$ so $b'[e_2'/x] \leq t'$. Consequently $a \leq^* t'$, as desired.

Case 2: Suppose $t \equiv \mathit{case}(e, x.e_1, x.e_2)$, $t' \equiv \mathit{case}(e', x.e_1', x.e_2')$ and a are closed. Suppose further that $t \Downarrow^k a$, $e \leq^* e'$, $e_i \leq^* e_i'$ and the induction hypothesis holds. Then $e \Downarrow^{<k} \mathit{inj}_j(b)$ (for some $j = 1$ or 2) and $e_j[b/x] \Downarrow^{<k} a$. By the induction hypothesis, $\mathit{inj}_j(b) \leq^* e'$. By Howe's Lemma 2, there exists $b' \geq^* b$ such that $e' \Downarrow \mathit{inj}_j(b')$. By Howe's Lemma 1, $e_j[b/x] \leq^* e_j'[b'/x]$. By the induction hypothesis, $a \leq^* e_j'[b'/x]$. However, $t' \mapsto^* e_j'[v'/x]$ so $e_j'[v'/x] \leq t'$. Consequently $a \leq^* t'$, as desired.

Case 3: Suppose $t \equiv \pi_i(e)$, $t' \equiv \pi_i(e')$ and a are closed. Suppose further that $t \Downarrow^k a$, $e \leq^* e'$ and the induction hypothesis holds. Then $e \Downarrow^{<k} \langle b_1, b_2 \rangle$ and $b_i \Downarrow^{<k} a$. By the induction hypothesis, $\langle b_1, b_2 \rangle \leq^* e'$. By Howe's Lemma 2, there exist $b_1' \geq^* b_1$ and $b_2' \geq^* b_2$ such that $e' \Downarrow \langle b_1', b_2' \rangle$. By the induction hypothesis, $a \leq^* b_i'$. However, $t' \mapsto b_i'$ so $a \leq^* t'$.

Case 4: Suppose $t \equiv (e_1 =_A e_2)$, $t' \equiv (e_1' =_A e_2')$ and a are closed. Suppose further that $t \Downarrow^k a$, $e_i \leq^* e_i'$ and the induction hypothesis holds. Then, for some atoms α_1 and α_2, $e_i \Downarrow^{<k} \alpha_i$. If $\alpha_1 = \alpha_2$ then $a \equiv \mathit{true}$ and otherwise $a \equiv \mathit{false}$. By the induction hypothesis, $\alpha_i \leq^* e_i'$. By Howe's Lemma 2, $e_i' \Downarrow \alpha_i$. Hence $t' \Downarrow a$, so $a \leq^* t'$. The cases for $=_Z$, $\leq_Z$ and $\oplus$ are similar.

Case 5: Suppose $t \equiv \mathit{let}\ x = e_1\ \mathit{in}\ e_2$, $t' \equiv \mathit{let}\ x = e_1'\ \mathit{in}\ e_2'$ and a are closed. Suppose further that $t \Downarrow^k a$, $e_1 \leq^* e_1'$, $e_2 \leq e_2'$ and the induction hypothesis holds. Then $e_1 \Downarrow^{<k} v$ and $e_2[v/x] \Downarrow^{<k} a$. By the induction hypothesis, $v \leq^* e_1'$. By Howe's Lemma 2, there exists $v' \geq^* v$ such that $e_1' \Downarrow v'$. Then by Howe's Lemma 1, $e_2[v/x] \leq^* e_2'[v'/x]$. By the induction hypothesis, $a \leq^* e_2'[v'/x]$. However, $t' \mapsto^* e_2'[v'/x]$ so $a \leq^* t'$.

Case 6: Suppose $t \equiv \mathit{fix}(e)$, $t' \equiv \mathit{fix}(e')$ and a are closed. Suppose further that $t \Downarrow^k a$, $e \leq^* e'$ and the induction hypothesis holds. Then $e \Downarrow^{<k} \lambda x.b$ and $b[\mathit{fix}(e)/x] \Downarrow^{<k} a$. By the induction hypothesis, $\lambda x.b \leq^* e'$. By Howe's Lemma 2, there exists $b' \geq^* b$ such that $e' \Downarrow \lambda x.b'$. By Howe's Lemma 1, $b[\mathit{fix}(e)/x] \leq^* b'[\mathit{fix}(e')/x]$, since $\mathit{fix}(e) \leq^* \mathit{fix}(e')$. By the induction hypothesis, $a \leq^* b'[\mathit{fix}(e')/x]$. However, $t' \mapsto b'[\mathit{fix}(e')/x]$ so $a \leq^* t'$. □

Lemma 5.7 If $e_1[t/x] \mapsto e_2$, and $e_1[t/x]$ is closed, and t is closed and noncanonical, then either

- there exists e_2' such that for any closed t', $e_1[t'/x] \mapsto e_2'[t'/x]$, or
- there exist e_1' and t' such that $e_1 \equiv e_1'[x/y]$, $t \mapsto t'$ and for any closed t'', $e_1'[t'', t/x, y] \mapsto e_1'[t'', t'/x, y]$.

Proof

Suppose $e_1 \equiv x$. Then $t \equiv e_1[t/x] \mapsto e_2$. Let $e_1' = y$ and $t' = e_2$. Then $e_1'[t'', t/x, y] \equiv t \mapsto t' \equiv e_1'[t'', t'/x, y]$. The remaining cases are by induction on the derivation of $e_1[t/x] \mapsto e_2$. I show the lambda rules; the other cases are similar.

Suppose the rule used is:

$$\overline{(\lambda z.b)a \mapsto b[a/z]}$$

The term $e_1[t/x]$ must have the form of a lambda abstraction applied to an argument. Thus e_1 must be of the form $(\lambda z.b)a$, since $e_1 \equiv x$ is already handled and $e_1 \equiv x\, a$ is impossible because t is noncanonical. Let $e_2' = b[a/z]$ and suppose t' is closed. Then:

$$\begin{array}{rcll} e_1[t'/x] & \equiv & (\lambda z.b[t'/x])(a[t'/x]) & \\ & \mapsto & b[t'/x](a[t'/x]/z) & \\ & \equiv & b[a/z][t'/x] & \text{(since } t' \text{ is closed)} \\ & \equiv & e_2'[t'/x] & \end{array}$$

Suppose the rule used is:

$$\frac{f \mapsto f'}{f\,a \mapsto f'\,a}$$

Then it must be the case that e_1 is of the form $f_1\,a$ (since $e_1 \equiv x$ is already handled) and $f_1[t/x] \mapsto f_2$ (for some f_2). Hence the induction hypothesis holds for f_1. Suppose the first case holds: there exists f_2' such that for any closed t', $f_1[t'/x] \mapsto f_2'[t'/x]$. Let $e_2' = f_2'\,a$ and suppose t' is closed. Then:

$$\begin{array}{rcl} e_1[t'/x] & \equiv & (f_1[t'/x])(a[t'/x]) \\ & \mapsto & (f_2'[t'/x])(a[t'/x]) \\ & \equiv & e_2'[t'/x] \end{array}$$

Suppose the second case holds: there exist f_1' and t' such that $f_1 \equiv f_1'[x/y]$, $t \mapsto t'$ and for any closed t'', $f_1'[t'', t/x, y] \mapsto f_1'[t'', t'/x, y]$. Let $e_1' = f_1'\,a$ and suppose t'' is closed. Then:

$$\begin{array}{rcll} e_1'[t'', t/x, y] & \equiv & (f_1'[t'', t/x, y])(a[t'', t/x, y]) & \\ & \equiv & (f_1'[t'', t/x, y])(a[t'', t'/x, y]) & \text{(since } y \text{ is not free in } a) \\ & \mapsto & (f_1'[t'', t'/x, y])(a[t'', t'/x, y]) & \\ & \equiv & e_1'[t'', t'/x, y] & \end{array}$$

□

Lemma 5.8 For all f, e_1 and e_2 (where f is closed and x is the only free variable of e_1), there exist j and e_2' such that if $e_1[\mathit{fix}(f)/x] \mapsto^* e_2$ then $e_2 \equiv e_2'[\mathit{fix}(f)/x]$ and for all $k \geq j$, $e_2'[f^{k-j}/x] \leq e_1[f^k/x]$.

Proof

I show the lemma for evaluations of length exactly one. The result then follows by induction on the length of the evaluation sequence, summing the numbers j.

Use Lemma 5.7. Suppose the first case holds: there exists e_2' such that for any closed t, $e_1[t/x] \mapsto e_2'[t/x]$. Then $e_1[\mathit{fix}(f)/x] \mapsto e_2'[\mathit{fix}(f)/x]$ and, for any k, $e_1[f^k/x] \mapsto e_2'[f^k/x]$. Thus $e_2'[f^{k-0}/x] \leq e_1[f^k/x]$. Suppose the second case holds: there exists e_1' such that $e_1 \equiv e_1'[x/y]$, and for any closed t, $e_1'[t, \mathit{fix}(f)/x, y] \mapsto e_1'[t, f(\mathit{fix}(f))/x, y]$. Let $e_2' = e_1'[f\,x/y]$. Then $e_1[\mathit{fix}(f)/x] \mapsto e_2'[\mathit{fix}(f)/x]$. Suppose $k \geq 1$, then $e_2'[f^{k-1}/x] \equiv e_1'[f^{k-1}, f^k/x, y] \leq e_1'[f^k, f^k/x, y] \equiv e_1[f^k/x]$. □

Theorem 5.9 For all f, t and e (where f is closed), if $\forall j.\, e[f^j/x] \leq t$, then $e[\mathit{fix}(f)/x] \leq t$.

Proof

By induction on l that for all f, t and e (where f is closed), $(\forall j.\, e[f^j/x] \leq t) \Rightarrow e[\mathit{fix}(f)/x] \leq_l t$. The result follows by the definition of $\leq$. The basis is trivial.

Assume the induction hypothesis for l and $\forall j.\, e[f^j/x] \leq t$. Let σ be a substitution such that $\sigma(e[\mathit{fix}(f)/x])$ and $\sigma(t)$ are closed and suppose, without loss of generality, that σ does not substitute for x. Then $\sigma(e[\mathit{fix}(f)/x]) \equiv \sigma(e)[\mathit{fix}(f)/x]$, $\sigma(e[f^j/x]) \equiv \sigma(e)[f^j/x]$ (for any j), and the only free variable of $\sigma(e)$ is x. Suppose $\sigma(e[\mathit{fix}(f)/x]) \Downarrow e'$. By Lemma 5.8, $e' \equiv e''[\mathit{fix}(f)/x]$ and, for some j and all $k \geq j$, $e''[f^{k-j}/x] \leq \sigma(e)[f^k/x]$. Then, by assumption and transitivity, $\forall k \geq j.\, e''[f^{k-j}/x] \leq \sigma(t)$. Therefore, changing variables to replace $k - j$ with k, $e''[f^k/x] \leq \sigma(t)$ for any k.

Let $e'' = \theta(\vec{x}_1.t_1, \ldots, \vec{x}_n.t_n)$ (and suppose, without loss of generality, that x does not appear in any $\vec{x}_i$). Then $\sigma(t) \Downarrow \theta(\vec{x}_1.t_1', \ldots, \vec{x}_n.t_n')$ where, for $1 \leq i \leq n$ and any k, $t_i[f^k/x] \leq t_i'$. By induction, $t_i[\mathit{fix}(f)/x] \leq_l t_i'$, for any i. Therefore $\sigma(e[\mathit{fix}(f)/x]) \leq_{l+1} \sigma(t)$ and hence $e[\mathit{fix}(f)/x] \leq_{l+1} t$. □

Lemma 5.18

- $\mathrm{Adm}(A + B \mid y : S)$ if $\forall s \in S.\,(A + B)[s/y]$ type and $\mathrm{Adm}(A \mid y : S)$ and $\mathrm{Adm}(B \mid y : S)$.
- $\mathrm{Adm}(\Sigma x{:}A.B \mid y : S)$ if $\forall s \in S.\,(\Sigma x{:}A.B)[s/y]$ type and $\Sigma y{:}S.A$ type and $\mathrm{Adm}(A \mid y : S)$ and $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathrm{Adm}(\mathit{Void} \mid y : S)$, $\mathrm{Adm}(\mathit{Atom} \mid y : S)$, $\mathrm{Adm}(\mathbb{Z} \mid y : S)$ and $\mathrm{Adm}(\mathbb{E} \mid y : S)$
- $\mathrm{Adm}(a_1 = a_2 \ in\ A \mid y : S)$ if $\forall s \in S.\,(a_1 = a_2 \ in\ A)[s/y]$ type and $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a_1 \leq a_2 \mid y : S)$
- $\mathrm{Adm}(\overline{A} \mid y : S)$ if $\forall s \in S.\,\overline{A}[s/y]$ type and $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a\ in!\ A \mid y : S)$ if $\forall s \in S.\,(a\ in!\ A)[s/y]$ type

Proof

I show the product and equality cases; the other cases are similar but easier.

Case 1: For the product case, let f, t, t' and e be arbitrary. Suppose $e^{[\omega]} \in S$ and j is such that $\forall k \geq j.\, e^{[k]} \in S \wedge t^{[k]} = t'^{[k]} \in (\Sigma x{:}A.B)[e^{[k]}/y]$. It is necessary to show that $t^{[\omega]} = t'^{[\omega]} \in (\Sigma x{:}A.B)[e^{[\omega]}/y]$. Since $e^{[\omega]} \in S$, it follows that $(\Sigma x{:}A.B)[e^{[\omega]}/y]$ type. Both $t^{[j]}$ and $t'^{[j]}$ converge to pairs, so, by Corollary 5.6, $t^{[\omega]} \Downarrow \langle a, b\rangle$ and $t'^{[\omega]} \Downarrow \langle a', b'\rangle$ for some terms a, b, a' and b'. To get that $b = b' \in B[e^{[\omega]}/y][a/x]$, it suffices to show that $\pi_2(t^{[\omega]}) = \pi_2(t'^{[\omega]}) \in B[e^{[\omega]}/y][\pi_1(t^{[\omega]})/x]$. Rearranging, it suffices to show equality in $B[\pi_1(z), \pi_2(z)/y, x][\langle e, \pi_1(t)\rangle^{[\omega]}/z]$.

Since $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$, it suffices to show that $\langle e, \pi_1(t)\rangle^{[\omega]} \in \Sigma y{:}S.A$ and $\forall k \geq j.\, \langle e, \pi_1(t)\rangle^{[k]} \in \Sigma y{:}S.A \wedge \pi_2(t^{[k]}) = \pi_2(t'^{[k]}) \in B[\pi_1(z), \pi_2(z)/y, x][\langle e, \pi_1(t)\rangle^{[k]}/z]$. The former will follow from $a \in A[e^{[\omega]}/y]$ and the supposition. The left half of the latter also follows from the supposition. Rearranging the right half, it suffices to show that $\pi_2(t^{[k]}) = \pi_2(t'^{[k]}) \in B[e^{[k]}/y][\pi_1(t^{[k]})/x]$, which follows from the supposition. The proof that $a = a' \in A[e^{[\omega]}/y]$ is similar but easier. Hence $t^{[\omega]} = t'^{[\omega]} \in (\Sigma x{:}A.B)[e^{[\omega]}/y]$.

Case 2: For the equality case, again let f, t, t' and e be arbitrary. Suppose $e^{[\omega]} \in S$ and j is such that $\forall k \geq j.\, e^{[k]} \in S \wedge t^{[k]} = t'^{[k]} \in (a_1 = a_2 \ in\ A)[e^{[k]}/y]$. It is necessary to show that $t^{[\omega]} = t'^{[\omega]} \in (a_1 = a_2 \ in\ A)[e^{[\omega]}/y]$. Since $e^{[\omega]} \in S$, it follows that $(a_1 = a_2 \ in\ A)[e^{[\omega]}/y]$ type. Both $t^{[j]}$ and $t'^{[j]}$ converge to $\star$, so, by Corollary 5.6, $t^{[\omega]}$ and $t'^{[\omega]}$ converge to $\star$. It remains to show that $a_1[e^{[\omega]}/y] = a_2[e^{[\omega]}/y] \in A[e^{[\omega]}/y]$. Since $\mathrm{Adm}(A \mid y : S)$, it suffices to show that $\forall k \geq j.\, e^{[k]} \in S \wedge a_1[e^{[k]}/y] = a_2[e^{[k]}/y] \in A[e^{[k]}/y]$. This follows since $(a_1 = a_2 \ in\ A)[e^{[k]}/y]$ is inhabited for all $k \geq j$. □

Lemma 5.20 $\mathrm{Adm}(\Pi x{:}A.B \mid y : S)$ if $\forall s \in S.\,(\Sigma x{:}A.B)[s/y]$ type and $\mathrm{WCoAdm}(A \mid y : S)$ and $\forall s \in S, a \in A[s/y].\, \mathrm{Adm}(B[a/x] \mid y : S)$

Proof

Let f, t, t' and e be arbitrary. Suppose $e^{[\omega]} \in S$ and j is such that $\forall k \geq j.\, e^{[k]} \in S \wedge t^{[k]} = t'^{[k]} \in (\Pi x{:}A.B)[e^{[k]}/y]$. I need to show that $t^{[\omega]} = t'^{[\omega]} \in (\Pi x{:}A.B)[e^{[\omega]}/y]$. Since $e^{[\omega]} \in S$, it follows that $(\Pi x{:}A.B)[e^{[\omega]}/y]$ type. Both $t^{[j]}$ and $t'^{[j]}$ converge to lambda abstractions, so, by Corollary 5.6, $t^{[\omega]} \Downarrow \lambda x.b$ and $t'^{[\omega]} \Downarrow \lambda x.b'$ for some terms b and b'. Suppose $a = a' \in A[e^{[\omega]}/y]$. To get that $b[a/x] = b'[a'/x] \in B[e^{[\omega]}, a/y, x]$ it suffices to show that $t^{[\omega]}a = t'^{[\omega]}a' \in B[e^{[\omega]}, a/y, x]$.

Since $e^{[\omega]} \in S$, it follows that $\mathrm{Adm}(B[a/x] \mid y : S)$. Therefore, it suffices to show that for some j' and all $k \geq j'$, $t^{[k]}a = t'^{[k]}a' \in B[e^{[k]}, a/y, x]$. Since $\mathrm{WCoAdm}(A \mid y : S)$, there exists j'' such that $\forall k \geq j''.\, a = a' \in A[e^{[k]}/y]$. Therefore $j' = \max(j, j'')$ suffices. □

Lemma 5.21

- $A + B$ is (weakly) coadmissible for y in S if $\forall s \in S.\, (A+B)[s/y]$ type and A and B are (weakly) coadmissible for y in S
- $\mathrm{WCoAdm}(\Sigma x{:}A.B \mid y : S)$ if $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ type and $\mathrm{WCoAdm}(A \mid y : S)$ and $\forall s \in S, a \in A[s/y].\, \mathrm{WCoAdm}(B[a/x] \mid y : S)$
- $\mathrm{CoAdm}(\Sigma x{:}A.B|y : S)$ if $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$type and $\Sigma y{:}S.A$type and $\mathrm{CoAdm}(A|y : S)$ and $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- *Void*, *Atom*, $\mathbb{Z}$ and $\mathbb{E}$ are strongly or weakly coadmissible for y in any S
- $\mathrm{CoAdm}(a_1 = a_2\ in\ A \mid y : S)$ if $\forall s \in S.\, (a_1 = a_2\ in\ A)[s/y]$ type and $\mathrm{CoAdm}(A \mid y : S)$
- $a_1 \leq a_2$ is strongly or weakly coadmissible for y in any S
- $\overline{A}$ is (weakly) coadmissible for y in S if $\forall s \in S.\, \overline{A}[s/y]$ type and A is (weakly) coadmissible for y is S
- $a\ in!\ A$ is strongly or weakly coadmissible for y in S if $\forall s \in S.\, (a\ in!\ A)[s/y]$ type

Proof

The proof is largely similar to the preceding proofs, but inverted. I show the proofs for full coadmissibility of products and partial types.

Case 1: For the product case, let f, t, t' and e be arbitrary. Suppose $e^{[\omega]} \in S$, j is such that $\forall k \geq j.\, e^{[k]} \in S$, and $t^{[\omega]} = t'^{[\omega]} \in (\Sigma x{:}A.B)[e^{[\omega]}/y]$. It is necessary to show that there exists j' such that $\forall k \geq j'.\, t^{[k]} = t'^{[k]} \in (\Sigma x{:}A.B)[e^{[k]}/y]$. For any $k \geq j$, $(\Sigma x{:}A.B)[e^{[k]}/y]$ type. Both $t^{[\omega]}$ and $t'^{[\omega]}$ converge to pairs, so, by compactness, there exists some j'' such that for all $k \geq j''$, $t^{[k]}$ and $t^{[k]}$ converge to pairs. Thus it suffices to show that for some $j' \geq \max(j, j'')$ and all $k \geq j'$, $\pi_1(t^{[k]}) = \pi_1(t'^{[k]}) \in A[e^{[k]}/y]$ and $\pi_2(t^{[k]}) = \pi_2(t'^{[k]}) \in B[e^{[k]}, \pi_1(t^{[k]})/y, x]$. I show the latter; the former is similar.

Rearranging, it suffices to show equality in $B[\pi_1(z), \pi_2(z)/y, x][\langle e, \pi_1(t)\rangle^{[k]}/z]$. By coadmissibility, it suffices to show $\langle e, \pi_1(t)\rangle^{[\omega]} \in \Sigma y{:}S.A$ and $\exists j'''.\, \forall k \geq j'''.\, \langle e, \pi_1(t)\rangle^{[k]} \in \Sigma y{:}S.A$ and $\pi_2(t^{[\omega]}) = \pi_2(t'^{[\omega]}) \in B[\pi_1(z), \pi_2(z)/y, x][\langle e, \pi_1(t)\rangle^{[\omega]}/z]$. The first follows from the supposition and the second will follow from $\pi_1(t^{[k]}) \in A[e^{[k]}/y]$ and the supposition. Rearranging the third, is suffices to show that $\pi_2(t^{[\omega]}) = \pi_2(t'^{[\omega]}) \in B[e^{[\omega]}/y][\pi_1(t^{[\omega]})/x]$, which follows from the supposition.

Case 2: For the partial type case, let f, t, t' and e be arbitrary. Suppose $e^{[\omega]} \in S$, j is such that $\forall k \geq j.\, e^{[k]} \in S$, and $t^{[\omega]} = t'^{[\omega]} \in \overline{A}[e^{[\omega]}/y]$. I need to show that there exists j' such that $\forall k \geq j'.\, t^{[k]} = t'^{[k]} \in \overline{A}[e^{[k]}/y]$. For any $k \geq j$, $\overline{A}[e^{[k]}/y]$ type. Suppose $t^{[\omega]}$ does not converge. (Note the non-constructivity of this argument.) Then $t'^{[\omega]}$ does not converge and neither does $t^{[k]}$ or $t'^{[k]}$ for any k (since $t^{[k]} \leq t^{[\omega]}$ and $t'^{[k]} \leq t'^{[k]}$ for all k). Thus for all $k \geq j$, $t^{[k]} = t'^{[k]} \in \overline{A}[e^{[k]}/y]$.

Suppose $t^{[\omega]}$ converges. Then $t^{[\omega]} = t'^{[\omega]} \in A[e^{[\omega]}/y]$. By coadmissibility, there exists j' such that $\forall k \geq j'.\, t^{[k]} = t'^{[k]} \in A[e^{[k]}/y]$. Hence $\forall k \geq \max(j, j').\, t^{[k]} = t'^{[k]} \in \overline{A}[e^{[k]}/y]$. □

Lemma 5.23 (Predicate-admissibility and weak and full coadmissibility of case analysis.)

Proof

The proof follows the same lines as those of Lemmas 5.18 and 5.21.

Lemma 5.27 The type $\{x : A \mid B\}$ is admissible if:

- $\mathrm{Adm}(A)$, and
- $\mathrm{Adm}(B \mid x : A)$, and
- there exists b such that $b[a/x] \in B[a/x]$ whenever $a \in A$ and $\exists b'.\, b' \in B[a/x]$.

Proof

Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \{x : A \mid B\}$. Since $\{x : A \mid B\}$ is inhabited it is a type. Since $\mathrm{Adm}(A)$, $t^{[\omega]} = t'^{[\omega]} \in A$. By set membership, for all $k \geq j$ there exists b' such that $b' \in B[t^{[k]}/x]$. Thus $b[t^{[k]}/x] \in B[t^{[k]}/x]$ for all $k \geq j$. Hence $b[t^{[\omega]}/x] \in B[t^{[\omega]}/x]$ follows by predicate-admissibility. $\square$

Lemma 5.29 (Predicate-admissibility of set types.)

Proof

The proof follows the same lines as Lemma 5.27.

Lemma 5.30 (Weak and full coadmissibility and monotonicity of set types.)

Proof

The proof follows the same lines as Lemma 5.21 and Proposition 5.26.

Lemma 5.31 (Admissibility, predicate-admissibility, weak and full coadmissibility and monotonicity of quotient types.)

Proof

The proof follows the same lines as the proofs for the set type.

Theorem 5.32 There does not exist an algorithm that computes an integer j such that $\forall k \geq j.\, t = t' \in \overline{T}[e^{[k]}/x]$, when given S, T, f, t, t', e and i such that:

- $\forall s \in S.\, \overline{T}[s/x]$ type
- $\mathrm{CoAdm}(T \mid x : S)$
- $e^{[\omega]} \in S$
- $\forall k \geq i.\, e^{[k]} \in S$
- $t = t' \in \overline{T}[e^{[\omega]}/x]$

Proof

Suppose such an algorithm exists. Let g be an arbitrary term that computes a total function on integers; that is, $g \in \mathbb{Z} \to \mathbb{Z}$. Given the algorithm, we may effectively determine whether g iterated on 1 ever computes 0, which is certainly undecidable. Let $f = \lambda h.\, \lambda n.\ \mathit{if}\ n =_Z 0\ \mathit{then}\ 0\ \mathit{else}\ h(gx)$ and let $h = \mathit{fix}(f)$. Note that $f \in \overline{\mathbb{Z} \to \overline{\mathbb{Z}}} \to \mathbb{Z} \to \overline{\mathbb{Z}}$ and $h \in \mathbb{Z} \to \overline{\mathbb{Z}}$. By construction, g iterated on 1 computes 0 if and only if $h(1)\!\downarrow$.

We will use the algorithm to determine an upper bound on the number of recursive calls needed to simulate h. Let

$$\begin{array}{rcl} S & = & \overline{\mathbb{Z}} \\ T & = & x\ \mathit{in}!\ \mathbb{Z} \\ f & = & \text{as above} \\ t, t' & = & \mathit{let}\ \ y = h(1)\ \mathit{in}\ \star \\ e & = & w(1) \\ i & = & 0 \end{array}$$

Observe that $e^{[\omega]} = h(1)$ and $e^{[k]} = (f^k)(1)$, so the first four preconditions of the algorithm are satisfied. Moreover, $\mathrm{CoAdm}(T \mid x : S)$ can be shown constructively. For the final precondition, suppose $t\!\downarrow$. Then $h(1)\!\downarrow$ so $t \in h(1)\ \mathit{in}!\ \mathbb{Z}$.

Therefore let j be the result computed by the algorithm. I show that $f^j(1)\!\downarrow$ exactly when $h(1)\!\downarrow$. Since $f^j(1)$ approximates $h(1)$, it follows that $f^j(1)\!\downarrow$ implies $h(1)\!\downarrow$. By the algorithm specification, $t \in \overline{f^j(1)\ \mathit{in}!\ \mathbb{Z}}$. If $h(1)\!\downarrow$ then $t\!\downarrow$, so $t \in f^j(1)\ \mathit{in}!\ \mathbb{Z}$ and consequently $f^j(1)\!\downarrow$.

Let $h' = \overbrace{f(f \cdots f}^{j\ \text{times}}(\lambda y.1)\cdots)$, and observe that $f^j(1)\!\downarrow$ exactly when $h'(1) \Downarrow 0$. Consequently $h(1)\!\downarrow$ exactly when $h'(1) \Downarrow 0$. However, h' is total, so we may decide whether $h(1)\!\downarrow$ by running $h'(1)$. $\square$

Bibliography

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Twenty-Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 296–409, St. Petersburg, Florida, January 1996.

[4] Peter Aczel. Frege structures revisited. In B. Nordström and J. Smith, editors, *Proceedings of the 1983 Marstrand Workshop*, 1983.

[5] Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In *Second IEEE Symposium on Logic in Computer Science*, pages 215–221, Ithaca, New York, June 1987.

[6] Stuart Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[7] Philippe Audebaud. Partial objects in the calculus of constructions. In *Sixth IEEE Symposium on Logic in Computer Science*, pages 86–95, Amsterdam, July 1991.

[8] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.

[9] David A. Basin. An environment for automated reasoning about partial functions. In *Ninth International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[10] Michael Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.

[11] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*, 1997.

[12] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[13] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[14] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, Sendai, Japan, September 1997.

[15] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[16] Luca Cardelli. Phase distinctions in type theory. Unpublished manuscript, January 1988.

[17] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 70–79, San Diego, January 1988.

[18] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts*. Springer-Verlag, 1991.

[19] 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.

[20] Robert L. Constable. Intensional analysis of functions and types. Technical Report CSR-118-82, Department of Computer Science, University of Edinburgh, June 1982.

[21] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Topics in the Theory of Computation*, volume 24 of *Annals of Discrete Mathematics*, pages 21–37. Elsevier, 1985. Selected papers of the International Conference on Foundations of Computation Theory 1983.

[22] Robert L. Constable. Type theory as a foundation for computer science. In *Theoretical Aspects of Computer Software 1991*, volume 526 of *Lecture Notes in Computer Science*, pages 226–243, Sendai, Japan, 1991. Springer-Verlag.

[23] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. Technical report, Department of Computer Science, Cornell University, 1997.

[24] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Second IEEE Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.

[25] Robert L. Constable and Scott Fraser Smith. Computational foundations of basic recursive function theory. In *Third IEEE Symposium on Logic in Computer Science*, pages 360–371, Edinburgh, Scotland, July 1988.

[26] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[27] Thierry Coquand. An analysis of Girard's paradox. In *First IEEE Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, June 1986.

[28] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *The APIC Series*, pages 91–122. Academic Press, 1990.

[29] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[30] Karl Crary. Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, June 1997.

[31] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report TR98-1675, Department of Computer Science, Cornell University, April 1998.

[32] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[33] Michael Dummett. *Elements of Intuitionism.* Oxford Logic Guides. Clarendon Press, 1977.

[34] Andrzej Filinski. *Controlling Effects.* Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1996.

[35] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[36] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. dissertation, Université Paris VII, 1972.

[37] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1988.

[38] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[39] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[40] Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14:71–84, 1992.

[41] Robert Harper. Personal communication, 1993.

[42] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 93.

[43] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 206–219, January 1993.

[44] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[45] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[46] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[47] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[48] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998. To appear.

[49] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996.

[50] Jason J. Hickey. A semantics of objects in type theory. Unpublished manuscript, 1997.

[51] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[52] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Second IEEE Symposium on Logic in Computer Science*, pages 205–214, Ithaca, New York, June 1987.

[53] Douglas J. Howe. Equality in lazy computation systems. In *Fourth IEEE Symposium on Logic in Computer Science*, 1989.

[54] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. Technical report, Bell Labs, 1996.

[55] Shigeru Igarashi. Admissibility of fixed-point induction in first-order logic of typed theories. Technical Report AIM-168, Computer Science Department, Stanford University, May 1972.

[56] Paul Jackson. *The Nuprl Proof Development System, Version 4.1.* Department of Computer Science, Cornell University, 1994.

[57] Paul Bernard Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, January 1995.

[58] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 80–87, San Diego, January 1988.

[59] Christoph Kreitz. Formal reasoning about communications systems I. Technical report, Department of Computer Science, Cornell University, 1997.

[60] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[61] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[62] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[63] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[64] Xavier Leroy. *The Objective Caml System, Release 1.00.* Institut National de Recherche en Informatique et Automatique (INRIA), 1996.

[65] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, January 1986.

[66] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[67] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[68] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 1982.

[69] Paul Francis Mendler. *Inductive Definition in Type Theory.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[70] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type. In *Thirteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg Beach, Florida, January 1986.

[71] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[72] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[73] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[74] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[75] Greg Morrisett. *Compiling with Types.* Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[76] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, March 1998.

[77] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[78] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Sixth International Symposium on Programming*, number 167 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, April 1984.

[79] Chris Okasaki. Catenable double-ended queues. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 66–74, Amsterdam, June 1997.

[80] Erik Palmgren. An information system interpretation of Martin-Löf's partial type theory with universes. *Information and Computation*, 106:26–60, 1993.

[81] Erik Palmgren and Viggo Stoltenberg-Hansen. Domain interpretations of intuitionistic type theory. U.U.D.M. Report 1989:1, Uppsala University, Department of Mathematics, January 1989.

[82] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Twentieth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[83] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[84] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *Theoretical Aspects of Computer Software 1994*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346, Sendai, Japan, 1994. Springer-Verlag.

[85] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Twenty-Fourth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 40–53, Paris, January 1997.

[86] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland.

[87] Adrian Rezus. Semantics of constructive type theory. Technical Report 70, Informatics Department, Faculty of Science, Nijmegen, University, The Netherlands, September 1985.

[88] Dana Scott. Outline of a mathematical theory of computation. In *Fourth Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.

[89] Dana Scott. Lattice theoretic models for various type-free calculi. In *Fourth International Congress of Logic, Methodology and Philosophy of Science*. North-Holland, 1972.

[90] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

[91] Scott F. Smith. Hybrid partial-total type theory. *International Journal of Foundations of Computer Science*, 6:235–263, 1995.

[92] Scott Fraser Smith. *Partial Objects in Type Theory.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, January 1989.

[93] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[94] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, volume 2, pages 461–493. Cambridge University Press, 1992.

[95] Philip Wadler. The essence of functional programming. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[96] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, 2nd edition, 1910.