Programming Language Semantics in Foundational Type Theory

Karl Crary*
Cornell University

February 28, 1998

Abstract

There are compelling benefits to using foundational type theory as a framework for programming language semantics. I give a semantics of an expressive programming calculus in the foundational type theory of Nuprl. Previous type-theoretic semantics have used less expressive type theories, or have sacrificed important programming constructs such as recursion and modules. The primary mechanisms of this semantics for the core calculus are *partial types,* for typing recursion, *set types,* for encoding power and singleton kinds, which are used for subtyping and module programming, and *very dependent function types,* for encoding signatures. I then extend the semantics to modules using *phase-splitting.*

1 Introduction

Type theory has become a popular framework for formal reasoning in computer science [7, 38, 21] and has formed the basis for a number of automated deduction systems, including Automath, Nuprl, HOL and Coq [18, 5, 22, 2], 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 [48] and Harper and Mitchell [26], 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 [29] have given such an interpretation of full Standard ML (Revised) [42]. 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 [34] gave an embedding of a fragment of Objective CAML [37] into the foundational type theory of Nuprl. However, this fragment omitted some important constructs, such as recursion and modules.

*This material is based on work supported in part by ARPA/AF grant F30602-95-1-0047, NSF grant CCR-9244739, and AASERT grant N00014-95-1-0985. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

The difficulty has been that the same features of foundational type theories that make them so expressive also restrict 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 paper I show how to give a semantics to practical programming languages in foundational type theory. In particular, I give an embedding of a small but expressive programming language into a Martin-Löf-style 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 4.5. 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 [52, 43, 36, 24, 44]. The process of type-directed compilation consists (in part) of 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 [1, 23], set theory [33] and domain theory [49, 46, 45]. Thus, foundational type theory may be used as a "semantic intermediate language."

The paper is organized as follows: Section 2 presents the paper's object language, λ^K. This object language is a small programming calculus, not a practical programming language, so a formal elaborator must be invoked to relate these results to a full programming language. I do not present such an elaborator in this paper, but see Harper and Stone [29, 28] for a presentation of such an elaborator. Section 3 contains an overview of Nuprl, the foundational type theory I use in this paper. Section 4 contains the embedding that is the central technical contribution of the paper. Section 5 discusses promising directions for future work. Finally, Section 6 contains brief concluding remarks.

2 The Lambda-K Programming Calculus

As a case study to illustrate my technique, I use a predicative variant of λ^K, the high-level typed intermediate language in the KML compiler [17]. In this section I discuss λ^K. In the interest of brevity, the discussion assumes knowledge of several well-known programming constructs. Further discussion of the design of λ^K appears in Crary [17].

The syntax rules of λ^K appear in Figure 1. The overall structure of the core calculus (that is, λ^K minus modules and signatures) is similar to the higher-order polymorphic lambda calculus [19, 20] augmented with records at the term and type constructor level (and their corresponding types and kinds), and a fixpoint operator at the term level. In addition to the kind *Type*, the kind level also includes, for any type τ, the power kind $\mathcal{P}(\tau)$, which includes all subtypes of τ, and the singleton

kinds	κ	::=	$Type_i \mid \Pi\alpha{:}\kappa_1.\kappa_2 \mid \mathcal{P}_i(c) \mid \mathcal{S}_i(c) \mid \{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}$
constructors	c	::=	$\alpha \mid \lambda\alpha{:}\kappa.c \mid c_1[c_2] \mid \{\ell_1 = c_1, \ldots, \ell_n = c_n\} \mid \pi_\ell(c) \mid c_1 \rightarrow c_2 \mid c_1 \Rightarrow c_2 \mid$ $\forall\alpha{:}\kappa.c \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \mid ext(m)$
terms	e	::=	$x \mid \lambda x{:}c.e \mid e_1 e_2 \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid \pi_\ell(e) \mid fix_c(e) \mid ext(m)$
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 1: λ^K Syntax

kind $\mathcal{S}(\tau)$, which includes only τ. The kind level also contains the dependent function kind $\Pi\alpha{:}\kappa_1.\kappa_2$ and the dependent record kind $\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}$ where each ℓ_i is an external name (or label) and each α_i is an internal name (or binding occurrence; see Harper and Lillibridge [25] for discussion of internal and external names). Evaluation is intended to be call-by-value. The type level includes a type constructor $\Rightarrow$ for total functions and polymorphic functions are also required to be total.[1] Expressions that differ only by alpha-variation are considered to be identical, as are expressions that differ only by permuting fields in a record (so long as no field of a dependent record kind is permuted to before a field on which it depends).

To make this calculus predicative, the type-oriented kinds have level annotations i (i.e., $Type_i$, $\mathcal{P}_i(\tau)$ and $\mathcal{S}_i(\tau)$), which are integers ≥ 1. Each kind contains only types whose levels are strictly less than the given annotation, where the level of a type is the highest level annotation used within it. For $\mathcal{P}_i(\tau)$ or $\mathcal{S}_i(\tau)$ to be well-formed, the level of τ must be less than i. This mechanism is somewhat awkward, and is used to allow the calculus to be embedded in a predicative type theory. Section 5 contains some discussion of alternatives.

The static semantics of core λ^K is given by four judgements (inference rules appear in Appendix A). The subkinding judgement $\Gamma \vdash_K \kappa_1 \sqsubseteq \kappa_2$ indicates that (in context Γ) every type constructor in κ_1 is in κ_2. 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. The typing judgement $\Gamma \vdash_K e : c$ indicates that the term e has type c. Finally, the valuability judgement [29] $\Gamma \vdash_K e \downarrow c$ indicates that the term e has type c and evaluates without computational effects (in this setting this means just that it terminates).

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 mentioned above. A signature form for *total* functors (analogous to the total function type) would be straightforward to add, but is omitted for the sake of simplicity. 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

[1] Partial polymorphic functions could easily be added, but seem rarely to be useful in practice [53].

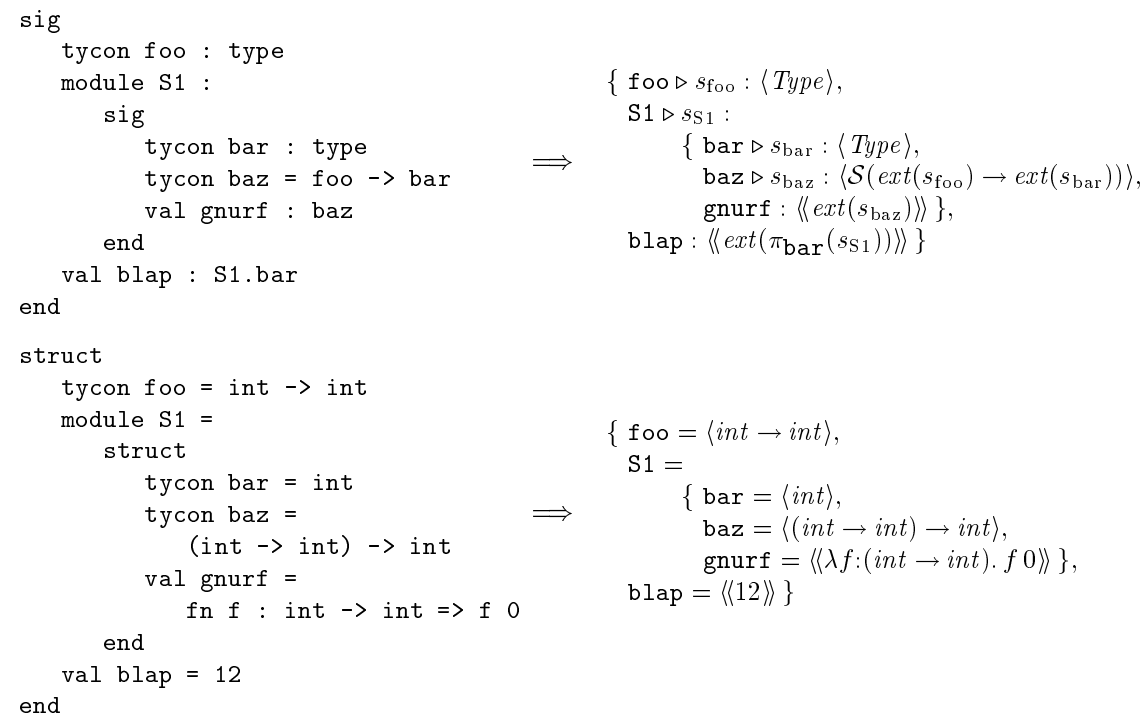

Figure 2: A Typical Module Encoding

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: $ext(m)$ for module m. If m has signature $\langle\kappa\rangle$, then the constructor $ext(m)$ has kind κ. Likewise, if m has signature $\langle\!\langle\tau\rangle\!\rangle$ then the term $ext(m)$ has type τ.

With the above constructs, it is easy to encode source-level modules. A 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 shows the encoding of a typical 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.

The static semantics of the module calculus is given by three additional judgements (inference rules appear in Appendix A). The subsignature judgement $\Gamma \vdash_K \sigma_1 \sqsubseteq \sigma_2$ indicates that (in context Γ) every module in σ_1 is in σ_2. The module similarity judgement $\Gamma \vdash_K m_1 \approx m_2 : \sigma$ indicates that m_1 and m_2 are members of σ and the portions of m_1 and m_2 that relate to types are equal (as members of σ). When m_1 and m_2 are the same, this judgement serves as a signature assignment judgement. Finally, the module valuability judgement $\Gamma \vdash_K m \downarrow \sigma$ indicates that m has signature σ and evaluates without computational effects.

	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 ($=_A$)
top	Top		

Figure 3: Type Theory Syntax

3 The Language of Type Theory

The type theory I use in this paper is the Martin-Löf-style type theory of Nuprl. A thorough discussion of Nuprl is beyond the scope of this paper, so the intent of this section is to give an overview of the programming features of 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 [31], but this will not be critical to our purposes. Detailed discussions of type theory, including the logic of types, appear in Martin-Löf [38, 39] and Constable [7, 8], and Nuprl specifically is discussed in Constable *et al.* [5]. As in the previous section, the discussion here assumes knowledge of several well-known programming constructs.

As base types, the theory contains integers (denoted by $\mathbb{Z}$), booleans (denoted by $\mathbb{B}$), strings (denoted by $Atom$), and the trivial type Top (which 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 (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 [20, 12, 41, 32], 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 the constructions in this paper may also be used in a call-by-value type theory with little modification.

To state the soundness of the embedding, 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 $T_1 \sqsubseteq T_2$, 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. The reader is referred to Crary [17] for the inference rules for the $\vdash_\nu$ judgement. 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 type theoretic operators appearing in this paper are given in Appendix B.

4 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 5 through 9.

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 4.5 could be proven within the framework of type theory. For simplicity, however, I will state the embedding and theorems in metatheory.

4.1 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:

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

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 4.3.

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 4.3. 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 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 $\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \to \mathit{int}\}$, 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, and $\star$ is an arbitrary member of *Top*.

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

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

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

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

Note that this embedding validates the desired subtyping relationship on records. Since $\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \to \mathit{int}\} \sqsubseteq \{\mathtt{x} : \mathit{int}\}$, we would like the embedding to respect the subtyping relationship: $[\![\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \to \mathit{int}\}]\!] \sqsubseteq [\![\{\mathtt{x} : \mathit{int}\}]\!]$. 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, $\mathit{if}\ a = \mathtt{f}\ \mathit{then}\ \mathbb{Z}{\to}\mathbb{Z}\ \mathit{else}\ \mathit{Top}$, is a subtype of *Top*.

4.2 Embedding Recursion

The usual approach to typing recursion, and the one used in λ^K, is to add a *fix* construct with the typing rule:

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

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:

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

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* [10, 11, 51, 9, 17]. 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 $\overline{T}$ [51, 9] are different than those on T, so we can safely type *fix* with the rule:[2]

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

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:[3]

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

[2] The second subgoal, that the type T be *admissible*, is a technical condition related to the notion of admissibility in LCF. This condition is required because fixpoint induction can be derived from the recursive typing rule [51]. However, all the types used in the embedding in this paper are admissible, so I ignore the admissibility condition in this paper. Additional details appear in Smith [51] and Crary [16, 17].

[3] 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.

Note that, as desired, $[\![\tau_1 \Rightarrow \tau_2]\!] \sqsubseteq [\![\tau_1 \to \tau_2]\!]$, since $[\![\tau_2]\!] \sqsubseteq \overline{[\![\tau_2]\!]}$. 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 $([\![\tau_1]\!] \to \overline{[\![\tau_2]\!]}) \to \overline{[\![\tau_1]\!] \to \overline{[\![\tau_2]\!]}}$. This type does not quite fit the *fix* typing rule, which requires the domain type to be partial, so we must coerce $[\![f]\!]$ to a fixable type. We do this by eta-expanding $[\![f]\!]$ to gain access to its argument and then eta-expanding that argument:

$$\lambda g.[\![f]\!](\lambda x.g\ x) \in \overline{[\![\tau_1]\!] \to \overline{[\![\tau_2]\!]}} \to \overline{[\![\tau_1]\!] \to \overline{[\![\tau_2]\!]}}$$

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

$$[\![\mathit{fix}_c(e)]\!] \stackrel{\text{def}}{=} \mathit{fix}(\lambda g.[\![e]\!](\lambda x.g\ x))$$

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. (Polymorphic functions are unaffected because all type expressions converge (Corollary 4).) 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:

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

Application is then embedded in the expected way:

$$[\![e_1 e_2]\!] \stackrel{\text{def}}{=} \mathit{let}\ x = [\![e_2]\!]\ \mathit{in}\ [\![e_1]\!]\, x$$

A final issue arises in regard to records. In the embedding of Section 4.1, 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:

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

4.3 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* [6]. 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:[4]

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

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):

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

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 $\{\ell \triangleright \alpha : \mathit{Type}_1, \ell' \triangleright \alpha' : \mathcal{P}_1(\alpha)\}$. We might naively attempt to encode this like the non-dependent record type as

$$\begin{array}{l} \Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell\ \mathit{then}\ \mathbb{U}_1\ \mathit{else} \\ \qquad \mathit{if}\ a =_A \ell'\ \mathit{then}\ \{T : \mathbb{U}_1 \mid T \sqsubseteq \alpha \wedge \alpha\ \mathit{in}\ \mathbb{U}_1\}\ \mathit{else}\ \mathit{Top} \end{array} \qquad \text{(wrong)}$$

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

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

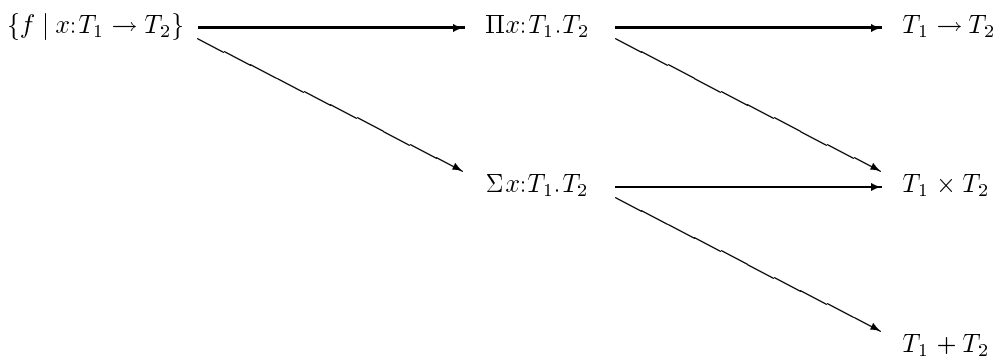

Figure 4: The Type Triangle

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!

The type I will use for this embedding is Hickey's *very dependent function type* [30]. 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 $T_2[t, g/x, f]$.[5]

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:

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

To understand where this type constructor fits in with the more familiar type constructors, consider the "type triangle" shown in Figure 4. 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 $\Sigma b{:}\mathbb{B}.\ \mathit{if}\ b\ \mathit{then}\ T_1\ \mathit{else}\ T_2$.[6]

[5] 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.

[6] 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.

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:

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

4.4 Embedding Modules

The interpretation of modules in λ^K is by phase-splitting, 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 into two parts. The technique used is derived from Harper, *et al.* [27].

The embeddings for modules and signatures are given by two syntax-directed mappings, $[\![\cdot]\!]_{\mathrm{c}}$ and $[\![\cdot]\!]_{\mathrm{r}}$, 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:

$$[\![m]\!] \quad \stackrel{\text{def}}{=} \quad \langle [\![m]\!]_{\mathrm{c}}, [\![m]\!]_{\mathrm{r}} \rangle$$

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:

$$[\![\sigma]\!] \quad \stackrel{\text{def}}{=} \quad \Sigma v{:}[\![\sigma]\!]_{\mathrm{c}}.[\![\sigma]\!]_{\mathrm{r}} v$$

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:

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

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.

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

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:

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

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):

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

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.

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

$$
\begin{array}{rcl}
[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
[\![\lambda\alpha{:}\kappa.c]\!] & \stackrel{\text{def}}{=} & \lambda\alpha.[\![c]\!] \\
[\![c_1[c_2]]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!][\![c_2]\!] \\
[\![\{\ell_1 = c_1, \ldots, \ell_n = c_n\}]\!] & \stackrel{\text{def}}{=} & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
 & & \quad \mathit{else}\ \star \\
 & & (\text{where } a \text{ does not appear free in } c_i) \\
[\![\pi_\ell(c)]\!] & \stackrel{\text{def}}{=} & [\![c]\!]\,\ell \\
[\![c_1 \to c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \to \overline{[\![c_2]\!]} \\
[\![c_1 \Rightarrow c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \to [\![c_2]\!] \\
[\![\forall\alpha{:}\kappa.c]\!] & \stackrel{\text{def}}{=} & \Pi\alpha{:}[\![\kappa]\!].[\![c]\!] \\
[\![\{\ell_1 : c_1, \ldots, \ell_n : c_n\}]\!] & \stackrel{\text{def}}{=} & \Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
 & & \quad \mathit{else}\ \mathit{Top} \\
 & & (\text{where } a \text{ does not appear free in } c_i)
\end{array}
$$

Figure 5: Embedding Types

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

Figure 6: Embedding Terms

$$
\begin{array}{rcl}
[\![\mathit{Type}_i]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T \ \mathit{admissible}\} \\
[\![\mathcal{P}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T \sqsubseteq [\![c]\!] \wedge [\![c]\!] \ \mathit{in}\ \mathbb{U}_i \wedge T \ \mathit{admissible}\} \\
 & & (\text{where } T \text{ does not appear free in } c) \\
[\![\mathcal{S}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T = [\![c]\!] \ \mathit{in}\ \mathbb{U}_i \wedge T \ \mathit{admissible}\} \\
 & & (\text{where } T \text{ does not appear free in } c) \\
[\![\Pi\alpha{:}\kappa_1.\kappa_2]\!] & \stackrel{\text{def}}{=} & \Pi\alpha{:}[\![\kappa_1]\!].[\![\kappa_2]\!] \\
[\![\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}]\!] & \stackrel{\text{def}}{=} & \{f \mid a{:}\mathit{Atom} \rightarrow \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\kappa_1]\!] \\
 & & \quad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\kappa_2]\!][f\,\ell_1/\alpha_1] \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad [\![\kappa_n]\!][f\,\ell_1 \cdots f\,\ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
 & & \quad \mathit{else}\ \mathit{Top}\} \\
 & & (\text{where } f, a \text{ do not appear free in } \kappa_i)
\end{array}
$$

Figure 7: Embedding Kinds

4.5 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 1:

Theorem 1 (Semantic Soundness) *For every λ^K context Γ, let $[\![\Gamma]\!]$ be defined as follows:*

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

Suppose $[\![\Gamma]\!] = v_1{:}T_1, \ldots, v_n{:}T_n$; then the following implications hold:

1. *If $\Gamma \vdash_K \kappa \sqsubseteq \kappa'$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu ([\![\kappa]\!]\ \mathit{in}\ \mathbb{U}_{level(\kappa)+1} \wedge [\![\kappa']\!]\ \mathit{in}\ \mathbb{U}_{level(\kappa')+1} \wedge [\![\kappa]\!] \sqsubseteq [\![\kappa']\!])$.*
2. *If $\Gamma \vdash_K c = c' : \kappa$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu [\![c]\!] = [\![c']\!]\ \mathit{in}\ [\![\kappa]\!]$.*
3. *If $\Gamma \vdash_K e : c$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu [\![e]\!]\ \mathit{in}\ \overline{[\![c]\!]}$.*
4. *If $\Gamma \vdash_K e \downarrow c$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu [\![e]\!]\ \mathit{in}\ [\![c]\!]$.*
5. *If $\Gamma \vdash_K \sigma \sqsubseteq \sigma'$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu ([\![\sigma]\!]\ \mathit{in}\ \mathbb{U}_{level(\sigma)+1} \wedge [\![\sigma']\!]\ \mathit{in}\ \mathbb{U}_{level(\sigma')+1} \wedge [\![\sigma]\!] \sqsubseteq [\![\sigma']\!])$.*
6. *If $\Gamma \vdash_K m \approx m' : \sigma$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu ([\![m]\!]\ \mathit{in}\ \overline{[\![\sigma]\!]} \wedge [\![m']\!]\ \mathit{in}\ \overline{[\![\sigma]\!]} \wedge [\![m]\!]_{\mathrm{c}} = [\![m']\!]_{\mathrm{c}}\ \mathit{in}\ [\![\sigma]\!]_{\mathrm{c}})$.*
7. *If $\Gamma \vdash_K m \downarrow \sigma$ then $v_1{:}T_1, \ldots, v_n{:}T_n \vdash_\nu [\![m]\!]\ \mathit{in}\ [\![\sigma]\!]$.*

$$
\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}} \\
 & & \quad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{c}}[f\, \ell_1/s_{1\mathrm{c}}] \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad [\![\sigma_n]\!]_{\mathrm{c}}[f\, \ell_1 \cdots f\, \ell_{n-1}/s_{1\mathrm{c}} \cdots s_{n-1\mathrm{c}}] \\
 & & \quad \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) \\
 & & \quad \mathit{if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{r}}(v\, \ell_2)[v\, \ell_1/s_{1\mathrm{c}}] \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad [\![\sigma_n]\!]_{\mathrm{r}}(v\, \ell_n) \\
 & & \qquad\quad [v\, \ell_1 \cdots v\, \ell_{n-1}/s_{1\mathrm{c}} \cdots s_{n-1\mathrm{c}}] \\
 & & \quad \mathit{else}\ \mathit{Top}\} \\
 & & (\text{where } v, a \text{ do not appear free in } \sigma_i)
\end{array}
$$

Figure 8: Embedding Signatures

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

Figure 9: Embedding Modules

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. Appendix B 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) follows directly from soundness and type preservation of Nuprl (Proposition 2).

Proposition 2 *If* $\vdash_\nu t \in T$ *and* $t \mapsto^* t'$ *then* $t' \in T$.

Proof

Not difficult, but outside the scope of this paper (see Crary [17]).

Corollary 3 (Type Preservation) *If* $\vdash_K e : \tau$ *and* $[\![e]\!] \mapsto^* t$ *then* $t \in [\![\tau]\!]$.

Another consequence of the soundness theorem is that the phase distinction [4, 27] 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 4:

Corollary 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.

5 Directions for Future Investigation

One important avenue for future work is to extend the semantics in this paper to explain stateful computation. One promising device for doing this is to encode stateful computations as monads [47, 35], 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. Type theory may express recursive types with only positive occurrences, but to allow negative occurrences is an open problem.[7]

[7] See Birkedal and Harper [3] for a promising approach that may lead to a solution of this problem.

The other main problem arising with a monadic interpretation of state has to do with predicativity. If polymorphic functions may be placed into the store, every function type, when monadized to take the store as an argument, will have a level as high as the highest level appearing in the store. Consequently, those monadized function types will not be valid arguments to some type abstractions when they should be. The obvious solution to this problem is to restrict references to be built with level-1 (non-polymorphic) types only. A more general solution would be to use a type theory with impredicative features. In addition to solving this problem, this would also eliminate the need for level annotations in the source calculus. The type theory of Mendler [40] provides such impredicative features and is quite similar to Nuprl; I have not used that framework in this paper out of desire to use a simpler and more standard theory. The Calculus of Constructions [14, 13] also supplies impredicative features and could likely also support the semantics in this paper.

6 Conclusion

I have shown how to give a type-theoretic semantics to an expressive programming calculus. 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, Crary [15] used 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 resulted in a 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 [50]. 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 [1, 23, 17].

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. This paper illustrates how to formalize these approximations. This view not only helps to *explain* programming languages and their features, as I have done in this paper, but also provides a greater insight into how we can bring more of the expressiveness of type theory into programming languages.

Acknowledgements

I am grateful to Robert Harper and Robert Constable for their many substantially helpful comments and suggestions.

References

[1] 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.

[2] 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.

[3] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*, 1997.

[4] Luca Cardelli. Phase distinctions in type theory. Manuscript, January 1988.

[5] 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.

[6] 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.

[7] 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.

[8] Robert L. Constable. Types in logic, mathematics and programming. In Sam Buss, editor, *Handbook of Proof Theory*, chapter 10. Elsevier Science, 1998.

[9] 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.

[10] 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.

[11] 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.

[12] Thierry Coquand. An analysis of Girard's paradox. In *First IEEE Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, June 1986.

[13] 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.

[14] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[15] 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.

[16] Karl Crary. Admissibility of fixpoint induction over partial types. Technical report, Department of Computer Science, Cornell University, 1998.

[17] Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1998. Forthcoming.

[18] 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.

[19] 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.

[20] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[21] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[22] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[23] Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14:71–84, 1992.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, School of Computer Science, June 1997.

[29] 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.

[30] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996.

[31] 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.

[32] 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.

[33] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. Technical report, Bell Labs, 1996.

[34] Christoph Kreitz. Formal reasoning about communications systems I. Technical report, Department of Computer Science, Cornell University, 1997.

[35] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[36] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[37] Xavier Leroy. *The Objective Caml System, Release 1.00*. Institut National de Recherche en Informatique et Automatique (INRIA), 1996. Available at http://pauillac.inria.fr/ocaml.

[38] 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.

[39] 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.

[40] Paul Francis Mendler. *Inductive Definition in Type Theory.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[41] 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.

[42] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[43] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[44] 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.

[45] Erik Palmgren. An information system interpretation of Martin-Löf's partial type theory with universes. *Information and Computation*, 106:26–60, 1993.

[46] 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.

[47] 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.

[48] 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.

[49] Adrian Rezus. Semantics of constructive type theory. Technical Report 70, Informatics Department, Faculty of Science, Nijmegen, University, The Netherlands, September 1985.

[50] 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.

[51] Scott Fraser Smith. *Partial Objects in Type Theory.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1989.

[52] 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.

[53] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

A Static Semantics of Lambda-K

$\Gamma \vdash \kappa_1 \sqsubseteq \kappa_2$		κ_1, κ_2 are valid kinds and κ_1 is a subkind of κ_2
$\Gamma \vdash_K c_1 = c_2 : \kappa$		c_1 and c_2 are equal in kind κ
$\Gamma \vdash_K e : c$		e has type c
$\Gamma \vdash_K e \downarrow c$		e is valuable and has type c
$\Gamma \vdash_K \sigma_1 \sqsubseteq \sigma_2$		σ_1, σ_2 are valid signatures and σ_1 is a subsignature of σ_2
$\Gamma \vdash_K m_1 \approx m_2 : \sigma$		m_1 and m_2 are similar in signature σ
$\Gamma \vdash_K m \downarrow \sigma$		m is valuable and has signature σ
$\Gamma \vdash_K \kappa$ kind	$\stackrel{\text{def}}{=}$	$\Delta \vdash_K \kappa \sqsubseteq \kappa$
$\Gamma \vdash_K c : \kappa$	$\stackrel{\text{def}}{=}$	$\Delta \vdash_K c = c : \kappa$
$\Gamma \vdash_K c_1 \sqsubseteq_i c_2$	$\stackrel{\text{def}}{=}$	$\Delta \vdash_K c_1 : \mathcal{P}_i(c_2)$
$\Gamma \vdash_K \sigma$ sig	$\stackrel{\text{def}}{=}$	$\Gamma \vdash_K \sigma \sqsubseteq \sigma$
$\Gamma \vdash_K m : \sigma$	$\stackrel{\text{def}}{=}$	$\Gamma \vdash_K m \approx m : \sigma$

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

A.1 Kind Formation and Subkinding

$$\frac{\Gamma \vdash_K \kappa_1 \sqsubseteq \kappa_2 \quad \Gamma \vdash_K \kappa_2 \sqsubseteq \kappa_3}{\Gamma \vdash_K \kappa_1 \sqsubseteq \kappa_3} \quad \text{(SubkindTrans)}$$

$$\frac{}{\Gamma \vdash_K \mathit{Type}_i \sqsubseteq \mathit{Type}_j} \; i \le j \quad \text{(Type)}$$

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

$$\frac{\begin{array}{l}\Gamma[\alpha_j : \kappa_j]^{[j=1\ldots i-1]} \vdash_K \kappa_i \sqsubseteq \kappa_i' \quad \text{for } 1 \le i \le m \\ \Gamma[\alpha_j : \kappa_j]^{[j=1\ldots i-1]} \vdash_K \kappa_i \text{ kind} \quad \text{for } m < i \le n \\ \Gamma[\alpha_j : \kappa_j']^{[j=1\ldots i-1]} \vdash_K \kappa_i' \text{ kind} \quad \text{for } 1 \le i \le m\end{array}}{\begin{array}{r}\Gamma \vdash_K \{\ell_i \triangleright \alpha_i : \kappa_i{}^{[i=1\ldots n]}\} \sqsubseteq \{\ell_i \triangleright \alpha_i : \kappa_i'{}^{[i=1\ldots m]}\} \\ \alpha_i^{[i=1\ldots m-1]} \notin \Gamma, n \ge m\end{array}} \quad \text{(DepRecord)}$$

$$\frac{\Gamma \vdash_K c_1 \sqsubseteq_i c_2}{\Gamma \vdash_K \mathcal{P}_i(c_1) \sqsubseteq \mathcal{P}_j(c_2)} \; i \le j \quad \text{(Pow)}$$

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

$$\frac{\Gamma \vdash_K c_1 = c_2 : \mathit{Type}_i}{\Gamma \vdash_K \mathcal{S}_i(c_1) \sqsubseteq \mathcal{S}_j(c_2)} \; i \le j \quad \text{(Sing)}$$

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

A.2 Kinding and Constructor Equality

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

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

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

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

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

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

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

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

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

A.3 Subtyping

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_K m_1 \approx m_2 : \langle\kappa\rangle}{\Gamma \vdash_K \mathit{ext}(m_1) = \mathit{ext}(m_2) : \kappa} \qquad \text{(TypeModElim)}$$

$$\frac{\Gamma \vdash_K c_1 \sqsubseteq_i c_2 \quad \Gamma \vdash_K c_2 \sqsubseteq_i c_3}{\Gamma \vdash_K c_1 \sqsubseteq_i c_3} \qquad \text{(SubtypeTrans)}$$

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

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

$$\frac{\Gamma \vdash_K c_1 : \mathit{Type}_i \quad \Gamma \vdash_K c_2 : \mathit{Type}_i}{\Gamma \vdash_K c_1 \Rightarrow c_2 \sqsubseteq_i c_1 \to c_2} \qquad \text{(TotalSub)}$$

$$\frac{\Gamma \vdash_K \kappa_1 \sqsubseteq \kappa_2 \quad \Gamma \vdash_K \kappa_2 \sqsubseteq \kappa_1 \quad \Gamma[\alpha:\kappa_1] \vdash_K c_1 \sqsubseteq_i c_2}{\Gamma \vdash_K \forall\alpha{:}\kappa_1.c_1 \sqsubseteq_i \forall\alpha{:}\kappa_2.c_2} \quad \alpha \notin \Gamma, \mathit{level}(\kappa_1) < i, \mathit{level}(\kappa_2) < i \qquad \text{(QuantSub)}$$

$$\frac{\Gamma \vdash_K c_i \sqsubseteq_j c_i' \quad \text{for } 1 \le i \le n}{\Gamma \vdash_K \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \sqsubseteq_j \{\ell_1 : c_1', \ldots, \ell_n : c_n'\}} \qquad \text{(RecordSub)}$$

$$\frac{\Gamma \vdash_K \{\ell_1 : c_1, \ldots, \ell_n : c_n\} : \mathit{Type}_i \quad \Gamma \vdash_K c : \mathit{Type}_i}{\Gamma \vdash_K \{\ell_1 : c_1, \ldots, \ell_n : c_n, \ell : c\} \sqsubseteq_i \{\ell_1 : c_1, \ldots, \ell_n : c_n\}} \qquad \text{(RecordFieldSub)}$$

A.4 Typing

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

$$\frac{\Gamma \vdash_K c_1 : \mathit{Type}_i \quad \Gamma[x : c_1] \vdash_K e : c_2}{\Gamma \vdash_K \lambda x{:}c_1.e : c_1 \to c_2} \ x \notin \Gamma \qquad \text{(ArrowIntro)}$$

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

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

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

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

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

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

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

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

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

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

A.5 Valuability

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

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

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

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

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

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

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

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

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

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

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

A.6 Signature Formation and Subsignatures

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

$$\frac{\Gamma \vdash_K \kappa_1 \sqsubseteq \kappa_2}{\Gamma \vdash_K \langle\kappa_1\rangle \sqsubseteq \langle\kappa_2\rangle} \qquad \text{(KindSig)}$$

$$\frac{\Gamma \vdash_K c_1 \sqsubseteq_i c_2}{\Gamma \vdash_K \langle\!\langle c_1 \rangle\!\rangle \sqsubseteq \langle\!\langle c_2 \rangle\!\rangle} \qquad \text{(TypeSig)}$$

$$\frac{\Gamma \vdash_K \sigma_1' \sqsubseteq \sigma_1 \quad \Gamma[s:\sigma_1'] \vdash_K \sigma_2 \sqsubseteq \sigma_2'}{\Gamma \vdash_K \Pi s{:}\sigma_1.\sigma_2 \sqsubseteq \Pi s{:}\sigma_1'.\sigma_2'} \qquad \text{(PiSig)}$$

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

A.7 Signature Assignment and Module Similarity

$$\frac{\Gamma \vdash_K m_2 \approx m_1 : \sigma}{\Gamma \vdash_K m_1 \approx m_2 : \sigma} \qquad \text{(SymmMod)}$$

$$\frac{\Gamma \vdash_K m_1 \approx m_2 : \sigma \quad \Gamma \vdash_K m_2 \approx m_3 : \sigma}{\Gamma \vdash_K m_1 \approx m_3 : \sigma} \qquad \text{(TransMod)}$$

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

$$\frac{\Gamma \vdash_K c_1 = c_2 : \kappa}{\Gamma \vdash_K \langle c_1 \rangle \approx \langle c_2 \rangle : \langle \kappa \rangle} \quad \text{(TypeModIntro)}$$

$$\frac{\Gamma \vdash_K e_1 : c \quad \Gamma \vdash_K e_2 : c}{\Gamma \vdash_K \langle\langle e_1 \rangle\rangle \approx \langle\langle e_2 \rangle\rangle : \langle\langle c \rangle\rangle} \quad \text{(TermModIntro)}$$

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

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

$$\frac{\Gamma \vdash_K m_i \approx m_i' : \sigma_i[m_j^{[j=1\ldots i-1]}/s_j^{[j=1\ldots i-1]}] \quad \text{for } 1 \le i \le n \qquad \Gamma \vdash_K \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\} \text{ sig}}{\Gamma \vdash_K \{\ell_i = m_i{}^{[i=1\ldots n]}\} \approx \{\ell_i = m_i'{}^{[i=1\ldots n]}\} : \{\ell_i \triangleright s_i : \sigma_i{}^{[i=1\ldots n]}\}} \quad \text{(RecordModIntro)}$$

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

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

A.8 Module Valuability

$$\frac{}{\Gamma \vdash_K s \downarrow \sigma} \; s : \sigma \in \Gamma \quad \text{(ModVarHalt)}$$

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

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

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

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

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

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

B Dynamic Semantics of Nuprl

A term is canonical if its top level form is any of the type constructors or introduction forms from Figure 3.

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

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

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

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}$$

$$\frac{}{(\lambda x.e_1)e_2 \mapsto e_1[e_2/x]}$$

$$\frac{e \mapsto e'}{\pi_1(e) \mapsto \pi_1(e')}$$

$$\frac{}{\pi_1(\langle e_1, e_2 \rangle) \mapsto e_1}$$

$$\frac{e \mapsto e'}{\pi_2(e) \mapsto \pi_2(e')}$$

$$\frac{}{\pi_2(\langle e_1, e_2 \rangle) \mapsto e_2}$$

$$\frac{e \mapsto e'}{if\ e\ then\ e_1\ else\ e_2 \mapsto if\ e'\ then\ e_1\ else\ e_2}$$

$$\frac{}{if\ true\ then\ e_1\ else\ e_2 \mapsto e_1}$$

$$\frac{}{if\ false\ then\ e_1\ else\ e_2 \mapsto e_2}$$

$$\frac{e_1 \mapsto e_1'}{e_1 =_A e_2 \mapsto e_1' =_A e_2}$$

$$\frac{e1 \text{ is a string literal} \quad e_2 \mapsto e_2'}{e_1 =_A e_2 \mapsto e_1 =_A e_2'}$$

$$\frac{e1 \text{ and } e2 \text{ are identical string literals}}{e_1 =_A e_2 \mapsto true}$$

$$\frac{e1 \text{ and } e2 \text{ are different string literals}}{e_1 =_A e_2 \mapsto false}$$

$$\frac{e_1 \mapsto e_1'}{let\ x = e_1\ in\ e_2 \mapsto let\ x = e_1'\ in\ e_2}$$

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

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