Programming Language Semantics in Foundational Type Theory

K. Crary
Cornell University
Ithaca, New York 14853
crary@cs.cornell.edu

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 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.

Keywords

Semantics, program verification, type theory, functional programming

1 INTRODUCTION

Type theory has become a popular framework for formal reasoning in computer science and has formed the basis for a number of automated deduction systems, including Automath, Nuprl, HOL and Coq, 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 (1981) and Harper and Mitchell (1993), 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 (1998) have given such an interpretation of full Standard ML (Revised) (Milner *et al.*, 1997). 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 (1997) gave an embedding of a fragment of Objective CAML into the foundational type theory of Nuprl. However, this fragment omitted some important constructs, such as recursion and modules.

The difficulty has been that the same features of foundational type theories that make them so expressive also 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 4.4. The usefulness of such semantics is also not limited to one particular programming language at a time. If two languages are given type-theoretic semantics, then one may use type theory to show relationships between the two, and when the semantics are simple, those relationships need be no more complicated than the inherent differences between the two. This is particularly useful in the area of type-directed compilation. 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 (Allen, 1987; Harper, 1992), set theory (Howe, 1996) and domain theory (Rezus, 1985; Palmgren and Stoltenberg-Hansen, 1989). 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

$$
\begin{array}{llll}
\textit{kinds} & \kappa & ::= & \mathit{Type}_i \mid \Pi\alpha{:}\kappa_1.\kappa_2 \mid \mathcal{P}_i(c) \mid \mathcal{S}_i(c) \mid \\
 & & & \quad \{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\} \\
\textit{constructors} & c & ::= & \alpha \mid \lambda\alpha{:}\kappa.c \mid c_1[c_2] \mid \{\ell_1 = c_1, \ldots, \ell_n = c_n\} \mid \\
 & & & \quad \pi_\ell(c) \mid c_1 \to c_2 \mid c_1 \Rightarrow c_2 \mid \\
 & & & \quad \forall\alpha{:}\kappa.c \mid \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \\
\textit{terms} & e & ::= & x \mid \lambda x{:}c.e \mid e_1 e_2 \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \\
 & & & \quad \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid \pi_\ell(e) \mid \mathit{fix}_c(e) \\
\textit{kind contexts} & \Delta & ::= & \bullet \mid \Delta[\alpha : \kappa] \\
\textit{type contexts} & \Gamma & ::= & \bullet \mid \Gamma[x : c]
\end{array}
$$

Figure 1 λ^K Syntax

elaborator in this paper, but see Harper and Stone (1998) 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. Due to space limitations, many technical details have been omitted; these may be found in the companion technical report (Crary, 1998b).

2 THE λ^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 (Crary, 1998c). In this section we discuss λ^K. In the interest of brevity, the discussion assumes knowledge of several well-known programming constructs.

The syntax rules of λ^K appear in Figure 1. The overall structure of the calculus is similar to the higher-order polymorphic lambda calculus (Girard, 1972) 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 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 (1994) 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.

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 λ^K is given by four judgements (details appear in the companion technical report). The subkinding judgement $\Delta \vdash_K \kappa_1 \sqsubseteq \kappa_2$ indicates that (in kind context Δ) every type constructor in κ_1 is in κ_2. The constructor equality judgement $\Delta \vdash_K c_1 = c_2 : \kappa$ indicates that c_1 and c_2 are equal as members of kind κ. The typing judgement $\Delta; \Gamma \vdash_K e : c$ indicates that (in kind context Δ and type context Γ) the term e has type c. Finally, the valuability judgement (Harper and Stone, 1998) $\Delta; \Gamma \vdash_K e \downarrow c$ indicates that the term e has type c and evaluates without computational effects (in this setting this means just that it terminates).

The λ^K calculus used in the KML compiler also includes operators for constructing higher-order modules similar to those of Harper and Lillibridge (1994). Space limitations prevent a discussion of those features here. However, much of the functionality of the module system is derived from the kind structure described above. Modules are discussed in detail in the companion technical report.

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 (Howard, 1980), but this will not be critical to our purposes. Detailed discussions of type theory, including the logic of types, appear in Martin-Löf (1982) and Constable (1991), and Nuprl specifically is discussed in Constable *et al.* (1986). 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

	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 2 Type Theory Syntax

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 (Girard, 1972), 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 $\mathit{if}\ b\ \mathit{then}\ \mathbb{Z}\ \mathit{else}\ \mathit{Top}$ (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 $t_1 = t_2\ \mathit{in}\ T$, 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 $t = t\ \mathit{in}\ T$. 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 2. The reader is referred to Crary (1998c) for their dynamic semantics and 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.

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 4, 5 and 6.

The embedding itself could be formulated in type theory, leaving to metatheory only the trivial task of encoding the abstract syntax of the programming language. Were this done, the theorems of Section 4.4 could be proven within the framework of type theory. For simplicity, however, I will state the embedding and theorems in metatheory.

4.1 Core Embedding

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

$$\begin{array}{rcl} [\![x]\!] & \stackrel{\text{def}}{=} & x \\ [\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\ [\![\lambda x{:}c.e]\!] & \stackrel{\text{def}}{=} & \lambda x.[\![e]\!] \\ [\![e_1 e_2]\!] & \stackrel{\text{def}}{=} & [\![e_1]\!][\![e_2]\!] \\ [\![c_1 \to c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \to [\![c_2]\!] \end{array} \tag{1}$$

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 $\forall\alpha{:}\kappa.c$:

$$\begin{aligned} [\![\Lambda\alpha{:}\kappa.e]\!] &\stackrel{\text{def}}{=} \lambda\alpha.[\![e]\!] \\ [\![e[c]]\!] &\stackrel{\text{def}}{=} [\![e]\!][\![c]\!] \\ [\![\forall\alpha{:}\kappa.c]\!] &\stackrel{\text{def}}{=} \Pi\alpha{:}[\![\kappa]\!].[\![c]\!] \end{aligned} \tag{2}$$

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:

$$[\![\mathit{Type}_i]\!] \stackrel{\text{def}}{=} \mathbb{U}_i \tag{3}$$

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

$$\lambda a.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ 1\ \mathit{else}\ \mathit{if}\ a =_A \mathtt{f}\ \mathit{then}\ \lambda x.x\ \mathit{else}\ \mathit{junk} \tag{4}$$

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

$$\Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ \mathbb{Z}\ \mathit{else}\ \mathit{if}\ a =_A \mathtt{f}\ \mathit{then}\ \mathbb{Z} \to \mathbb{Z}\ \mathit{else}\ \mathit{Top} \tag{5}$$

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

$$
\begin{array}{rcl}
[\![\{\ell_1 = e_1, \ldots, \ell_n = e_n\}]\!] & \stackrel{\text{def}}{=} & \lambda a.\, \textit{if } a =_A \ell_1 \textit{ then } [\![e_1]\!] \\
& & \qquad \vdots \\
& & \qquad \textit{else if } a =_A \ell_n \textit{ then } [\![e_n]\!] \\
& & \qquad \textit{else junk} \\
[\![\pi_\ell(e)]\!] & \stackrel{\text{def}}{=} & [\![e]\!]\,\ell \\
[\![\{\ell_1 : c_1, \ldots, \ell_n : c_n\}]\!] & \stackrel{\text{def}}{=} & \Pi a{:}Atom.\, \textit{if } a =_A \ell_1 \textit{ then } [\![c_1]\!] \\
& & \qquad\qquad \vdots \\
& & \qquad\qquad \textit{else if } a =_A \ell_n \textit{ then } [\![c_n]\!] \\
& & \qquad\qquad \textit{else Top}
\end{array}
\tag{6}
$$

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

$$
\frac{H \vdash_\nu e \in T \rightarrow T}{H \vdash_\nu \mathit{fix}(e) \in T} \tag{wrong}
$$

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:

$$
\frac{H \vdash_\nu e \in \mathit{Void}}{H \vdash_\nu e \in T} \tag{7}
$$

However, it would be easy, using *fix*, to derive a member of *Void*: the identity function can be given type $Void \rightarrow Void$, 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.

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* (Constable and Smith, 1987; Smith, 1989; Crary, 1998c). 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 $\overline{T}$ (Smith, 1989; Constable and Crary, 1997) are different than those on T, so we can safely type *fix* with the rule:*

$$\frac{H \vdash_\nu e \in \overline{T} \rightarrow \overline{T} \quad H \vdash_\nu T \ \mathit{admissible}}{H \vdash_\nu \mathit{fix}(e) \in \overline{T}} \tag{8}$$

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:†

$$\begin{array}{rcl} [\![c_1 \rightarrow c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \rightarrow \overline{[\![c_2]\!]} \\ [\![c_1 \Rightarrow c_2]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!] \rightarrow [\![c_2]\!] \\ [\![\forall \alpha{:}\kappa.c]\!] & \stackrel{\text{def}}{=} & (\alpha{:}[\![\kappa]\!]) \rightarrow [\![c]\!] \end{array} \tag{9}$$

Note that, as desired, $[\![\tau_1 \Rightarrow \tau_2]\!] \sqsubseteq [\![\tau_1 \rightarrow \tau_2]\!]$, 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 $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$, and suppose, for simplicity only, that f is valuable. Then $[\![f]\!]$

*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 (Smith, 1989). 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 Crary (1998a).

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

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

$$\lambda g.\llbracket f \rrbracket(\lambda x.g\,x) \in \overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}} \to \overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}} \tag{10}$$

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

$$\llbracket \mathit{fix}_c(e) \rrbracket \quad \stackrel{\text{def}}{=} \quad \mathit{fix}(\lambda g.\llbracket e \rrbracket(\lambda x.g\,x)) \tag{11}$$

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:

$$\frac{H \vdash_\nu e_1 \in \overline{T}_2 \quad H[x : T_2] \vdash_\nu e_2 \in \overline{T_1}}{H \vdash_\nu \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \in \overline{T_1}} \tag{12}$$

Application is then embedded in the expected way:

$$\llbracket e_1 e_2 \rrbracket \quad \stackrel{\text{def}}{=} \quad \mathit{let}\ x = \llbracket e_2 \rrbracket\ \mathit{in}\ \llbracket e_1 \rrbracket\, x \tag{13}$$

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:

$$\begin{array}{lll} \llbracket \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \rrbracket & \stackrel{\text{def}}{=} & \mathit{let}\ x_1 = \llbracket e_1 \rrbracket\ \mathit{in} \cdots \\ & & \mathit{let}\ x_n = \llbracket e_n \rrbracket\ \mathit{in} \\ & & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\ & & \quad \vdots \\ & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\ & & \quad \mathit{else\ junk} \end{array} \tag{14}$$

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* (Constable, 1985). 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:*

$$\begin{array}{lcl} [\![\mathcal{P}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T \sqsubseteq [\![c]\!] \wedge [\![c]\!] \; \mathit{in} \; \mathbb{U}_i\} \\ [\![\mathcal{S}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T = [\![c]\!] \; \mathit{in} \; \mathbb{U}_i\} \end{array} \tag{15}$$

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:

$$\begin{array}{rcl} [\![\lambda\alpha{:}\kappa.c]\!] & \stackrel{\text{def}}{=} & \lambda\alpha.[\![c]\!] \\ [\![c_1[c_2]]\!] & \stackrel{\text{def}}{=} & [\![c_1]\!][\![c_2]\!] \\ [\![\Pi\alpha{:}\kappa_1.\kappa_2]\!] & \stackrel{\text{def}}{=} & \Pi\alpha{:}[\![\kappa_1]\!].[\![\kappa_2]\!] \end{array} \tag{16}$$

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):

$$\begin{array}{rcl} [\![\{\ell_1 = c_1, \ldots, \ell_n = c_n\}]\!] & \stackrel{\text{def}}{=} & \lambda a.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\ & & \quad \vdots \\ & & \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\ & & \mathit{else\ junk} \\ [\![\pi_\ell(c)]\!] & \stackrel{\text{def}}{=} & [\![c]\!]\,\ell \end{array} \tag{17}$$

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 de-

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

pendent 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

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

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 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 a *very dependent function type* (Hickey, 1996). 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 $T_2[t, g/x, f]$.*

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:

$$\begin{aligned}\{f \mid a{:}\mathit{Atom} \to\ & \mathit{if}\ a =_A \ell\ \mathit{then}\ \mathbb{U}_1\ \mathit{else} \\ & \quad \mathit{if}\ a =_A \ell'\ \mathit{then}\ \{T : \mathbb{U}_1 \mid T \sqsubseteq f\,\ell \wedge f\,\ell\ \mathit{in}\ \mathbb{U}_1\}\ \mathit{else}\ \mathit{Top}\}\end{aligned} \qquad (18)$$

To understand where this type constructor fits in with the more familiar type constructors, consider the "type triangle" shown in Figure 3. 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

*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.

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

Figure 3 The Type Triangle

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 $\Sigma b{:}\mathbb{B}.\ \mathit{if}\ b\ \mathit{then}\ T_1\ \mathit{else}\ T_2$.

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:

$$\begin{array}{l}
[\![\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}]\!] \\
\quad \stackrel{\text{def}}{=} \quad \{f \mid a{:}Atom \to \begin{array}[t]{l}
\mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\kappa_1]\!] \\
\mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\kappa_2]\!][f\,\ell_1/\alpha_1] \\
\vdots \\
\mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
\qquad [\![\kappa_n]\!][f\,\ell_1 \cdots f\,\ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
\mathit{else}\ \mathit{Top}\}
\end{array}
\end{array} \tag{19}$$

4.4 Properties of the Embedding

I conclude my presentation of the type-theoretic semantics of λ^K by examining some of the important properties of the semantics. We want the embedding to validate the intuitive meaning of the judgements of λ^K's static semantics. If κ_1 is a subkind of κ_2 then we want the embedded kind $[\![\kappa_1]\!]$ to be a subtype

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

Figure 4 Embedding Types

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). This is stated in Theorem 1:

Theorem 1 (Semantic Soundness) *Suppose* $\Delta = [\alpha_1 : \kappa_1] \cdots [\alpha_n : \kappa_n]$ *and* $\Gamma = [x_1 : c_1] \cdots [x_m : c_m]$*, then the following implications hold:*

1. *If* $\Delta \vdash_K \kappa \sqsubseteq \kappa'$ *then* $\alpha_1{:}[\![\kappa_1]\!], \ldots, \alpha_n{:}[\![\kappa_n]\!] \vdash_\nu ([\![\kappa]\!] \sqsubseteq [\![\kappa']\!] \wedge [\![\kappa]\!] \in \mathbb{U}_{level(\kappa)+1} \wedge [\![\kappa']\!] \in \mathbb{U}_{level(\kappa')+1})$.
2. *If* $\Delta \vdash_K c = c' : \kappa$ *then* $\alpha_1{:}[\![\kappa_1]\!], \ldots, \alpha_n{:}[\![\kappa_n]\!] \vdash_\nu [\![c]\!] = [\![c']\!]$ *in* $[\![\kappa]\!]$.
3. *If* $\Delta; \Gamma \vdash_K e : c$ *then* $\alpha_1{:}[\![\kappa_1]\!], \ldots, \alpha_n{:}[\![\kappa_n]\!], x_1{:}[\![c_1]\!], \ldots, x_m{:}[\![c_m]\!] \vdash_\nu [\![e]\!] \in \overline{[\![c]\!]}$.
4. *If* $\Delta; \Gamma \vdash_K e \downarrow c$ *then* $\alpha_1{:}[\![\kappa_1]\!], \ldots, \alpha_n{:}[\![\kappa_n]\!], x_1{:}[\![c_1]\!], \ldots, x_m{:}[\![c_m]\!] \vdash_\nu [\![e]\!] \in [\![c]\!]$.

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

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

Figure 5 Embedding Terms

the type of a program. The evaluation of t in one step to t' is denoted by $t \mapsto t'$. 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 (1998c)).

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 (Harper *et al.*, 1990) 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

$$
\begin{array}{rcl}
[\![\mathit{Type}_i]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T \ \mathit{admissible}\} \\
[\![\mathcal{P}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T \sqsubseteq [\![c]\!] \wedge [\![c]\!] \ \mathit{in} \ \mathbb{U}_i \wedge T \ \mathit{admissible}\} \\
 & & \text{(where } T \text{ does not appear free in } c\text{)} \\
[\![\mathcal{S}_i(c)]\!] & \stackrel{\text{def}}{=} & \{T : \mathbb{U}_i \mid T = [\![c]\!] \ \mathit{in} \ \mathbb{U}_i \wedge T \ \mathit{admissible}\} \\
 & & \text{(where } T \text{ does not appear free in } c\text{)} \\
[\![\Pi\alpha{:}\kappa_1.\kappa_2]\!] & \stackrel{\text{def}}{=} & \Pi\alpha{:}[\![\kappa_1]\!].[\![\kappa_2]\!]
\end{array}
$$

$$
\begin{array}{l}
[\![\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}]\!] \\
\quad \stackrel{\text{def}}{=} \quad \{f \mid a{:}\mathit{Atom} \to \mathit{if} \ a =_A \ell_1 \ \mathit{then} \ [\![\kappa_1]\!] \\
\qquad\qquad\qquad\qquad\qquad \mathit{else} \ \mathit{if} \ a =_A \ell_2 \ \mathit{then} \ [\![\kappa_2]\!][f\,\ell_1/\alpha_1] \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad \mathit{else} \ \mathit{if} \ a =_A \ell_n \ \mathit{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![\kappa_n]\!][f\,\ell_1 \cdots f\,\ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
\qquad\qquad\qquad\qquad\qquad \mathit{else} \ \mathit{Top}\} \\
\text{(where } f, a \text{ do not appear free in } \kappa_i\text{)}
\end{array}
$$

Figure 6 Embedding Kinds

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 (Peyton Jones and Wadler, 1993), 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.*

The other main problem arising with a monadic interpretation of state has

*See Birkedal and Harper (1997) for a promising approach that may lead to a solution of this problem.

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 (1987) 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 (Coquand and Huet, 1988) also supplies impredicative features and could likely also support the semantics in this paper.

6 CONCLUSION

Aside from its advantages for formal program reasoning, 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 (1997) 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 (Scott and Stratchey, 1971). 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.

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.

REFERENCES

Allen, S. (1987) 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.

Birkedal, L. and Harper, R. (1997) Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*.

Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., and Smith, S. (1986) *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.

Constable, R. L. (1985) 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. Selected papers of the International Conference on Foundations of Computation Theory 1983.

Constable, R. L. (1991) 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. Springer-Verlag.

Constable, R. L. and Crary, K. (1997) Computational complexity and induction for partial computable functions in type theory. Technical report, Department of Computer Science, Cornell University.

Constable, R. L. and Smith, S. F. (1987) Partial objects in constructive type theory. In *Second IEEE Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York.

Coquand, T. and Huet, G. (1988) The calculus of constructions. *Information and Computation*, **76**:95–120.

Crary, K. (1997) Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam.

Crary, K. (1998a) Admissibility of fixpoint induction over partial types. Technical report, Department of Computer Science, Cornell University.

Crary, K. (1998b) Programming language semantics in foundational type theory. Technical Report TR98-1666, Department of Computer Science, Cornell University.

Crary, K. (1998c) *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York. Forthcoming.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII.

Harper, R. (1992) Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, **14**:71–84.

Harper, R. and Lillibridge, M. (1994) 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.

Harper, R. and Mitchell, J. C. (1993) On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, **15**(2):211–252.

Harper, R., Mitchell, J. C., and Moggi, E. (1990) Higher-order modules and the phase distinction. In *Seventeenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, San Francisco.

Harper, R. and Stone, C. (1998) A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press. To appear.

Hickey, J. J. (1996) Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*.

Howard, W. (1980) The formulas-as-types notion of construction. In Seldin, J. P. and Hindley, J. R., editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press.

Howe, D. J. (1996) Semantic foundations for embedding HOL in Nuprl. Technical report, Bell Labs.

Kreitz, C. (1997) Formal reasoning about communications systems I. Technical report, Department of Computer Science, Cornell University.

Martin-Löf, P. (1982) 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.

Mendler, P. F. (1987) *Inductive Definition in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997) *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts.

Palmgren, E. and Stoltenberg-Hansen, V. (1989) Domain interpretations of intuitionistic type theory. U.U.D.M. Report 1989:1, Uppsala University, Department of Mathematics.

Peyton Jones, S. L. and Wadler, P. (1993) Imperative functional programming. In *Twentieth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina.

Reynolds, J. C. (1981) The essence of Algol. In de Bakker, J. W. and van Vliet, J. C., editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam. North-Holland.

Rezus, A. (1985) Semantics of constructive type theory. Technical Report 70, Informatics Department, Faculty of Science, Nijmegen, University, The Netherlands.

Scott, D. and Strachey, C. (1971) 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.

Smith, S. F. (1989) *Partial Objects in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York.

BIOGRAPHY

Karl Crary earned his B.S. in Computer Science from Carnegie Mellon University in 1993 and is now a Ph.D. candidate at Cornell University. His main interests include programming language design and semantics, type theory, and typed compilation.