A Kind System for Typed Machine Language

Andrew W. Appel   Christopher D. Richards   Kedar N. Swadi

Princeton University, October 2002

{appel,richards,kswadi}@cs.princeton.edu

ABSTRACT

One of the aims of Foundational Proof-Carrying Code (FPCC) is to incorporate a completely semantic description of types into the Proof-Carrying Code framework. FPCC describes types as complex predicates, some of which require properties like *contractiveness*, *representability*, and *extensionality* to hold. We keep track of these properties by encoding them within kinds. In this paper, we give a syntactic kinding system with semantic proofs.

1. INTRODUCTION

Some of the early frameworks for Proof-Carrying Code (PCC) [12] assumed the soundness of the typing rules for a particular type system. Foundational Proof-Carrying Code (FPCC) [3] reduces the size of the Trusted Computing Base by giving semantics to types and instructions in terms of higher-order logic and arithmetic facts, and using these semantics in machine-checked proofs of the typing rules.

In the FPCC project at Princeton, we have a type system targeted towards compilation, with safety proofs, of languages such as Java and ML. Among the type operators in this system are the recursion operator rec, and the reference operators box (for immutable references) and ref (for mutable references). Recursive types are written as $\mathsf{rec}(f)$, where f is a type function. In the semantic systems of MacQueen, Plotkin, Sethi [9] and Appel and McAllester [5], the fold/unfold lemma to manipulate recursive types,

$$\mathsf{rec}\, f \;=\; f(\mathsf{rec}\, f),$$

is provable only if f is *contractive*. That is, f must be the composition of at least one contractive operator with other contractive and nonexpansive operators. As a result, it becomes necessary to keep track of the contractiveness and the nonexpansiveness of types to ensure well formedness.

To give a model for mutable references, we use the work of Ahmed et al. [2]. In this model, a reference type $\mathsf{ref}(\tau)$ is well formed if τ is *representable*. To model core ML, we often require a composition of operators like $\exists$, ref, and rec in ways which require us to keep track of properties like *contractiveness* as well as *representability*.

Finally, to model arithmetic dataflow, we use singleton integer types. When quantifying over such types in dataflow equations, we also wish to keep track of which types are singletons.

The previous approaches to semantics of recursive types explain how to prove the contractiveness of a single-argument type function, and how to compose single-argument contractive and nonexpansive type functions. Unfortunately, in the type systems we use in real life, type expressions often use many type variables at once. That is, deep inside a type expression such as

$$\Lambda\alpha.\ \mathsf{rec}\,\beta.\,\alpha\ \times\ \exists\gamma.\,(\gamma\times\beta)$$

there are subexpressions with several free variables. These are not simply compositions of one-argument type functions. What we need is an organizing principle for applying the semantic ideas for recursive types or mutable references to nontrivial type expressions.

In this paper, we present a kinding system to track these properties and ensure that all our types are well formed. Properties we wish to track are encoded into our kinds, the semantics of which are hidden in their definitions. Our kinding judgments are backed by machine-checked proofs; some of these proofs are complete and checked in the Twelf system; others we are still in the process of implementing from hand-verified arguments. The syntactic presentation also allows us to achieve modularity; higher layers in our system which use kinds and kinding judgments are shielded from their semantics. We expect any foundational semantic model for core ML to require addressing the properties listed above, and believe that the kinding system to be useful for all such models.

Our kinding system is restricted to first-order kinds. Since our current goal in the FPCC project is to be able to compile core ML, it is enough for us to restrict ourselves to first-order kinds.

We have type functions arising out of the uses of type operators like rec and $\exists$, and we use de Bruijn indices [7] with explicit substitutions [1] to manage applications of arguments to these type functions. Our de Bruijn numbering starts at $\underline{0}$. We do not use an explicit Λ anywhere; in our calculus, the arity of a type expression (possibly with free variables) is known from the context in which it appears, so the Λ is unnecessary.

Examples.

$\mathsf{ref}(\mathsf{int})$

A mutable reference to an integer.

$\mathsf{box}(\mathsf{const}(6))$

An immutable pointer to a value which must be the constant 6.

$\exists(\underline{0} \times \underline{0})$

A pair of values, both of which have the same type. In our system the cartesian product operator $\times$ is not a primitive, but is instead defined as the intersection of field types.

$\mathsf{field}(0, \mathsf{const}(3)) \cap \mathsf{field}(1, \mathsf{int}) \cap \mathsf{field}(2, \underline{0})$

A 3-tuple containing a constant integer 3, an integer, and a free type variable $\underline{0}$. Because this expression has a free type variable, it must be interpreted in some context that gives a binding for $\underline{0}$. Note that not even $\mathsf{field}(i, \tau)$ is primitive; it is defined as $\mathsf{offset}(i, \mathsf{box}(\tau))$. The use of simpler primitives gives the compiler greater flexibility in arranging data-structure layouts.

The above expression could also be considered as a function of its free de Bruijn index, as in

$$\lambda\alpha.(\mathsf{const}(3), \mathsf{int}, \alpha) \ ;$$

or even a function

$$\lambda\alpha.\lambda\beta.(\mathsf{const}(3), \mathsf{int}, \alpha)$$

where argument β fails to appear. In our calculus, the arity of an expression is determined by context.

$\mathsf{rec}(\mathsf{field}(0, \mathsf{const}(3)) \cap \mathsf{field}(1, \mathsf{int}) \cap \mathsf{field}(2, \underline{0}))$

An (infinite) list type; the type variable $\underline{0}$ is bound by the rec operator.

$(\mathsf{field}(0, \mathsf{const}(3)) \cap \mathsf{field}(1, \mathsf{int}) \cap \mathsf{field}(2, \underline{0}))[\mathsf{char} \cdot \mathsf{id}]$

This is a closed expression in which the explicit substitution $[\mathsf{char} \cdot \mathsf{id}]$ replaces $\underline{0}$ with the character type.

2. SEMANTIC MODEL OF TYPES

Our semantic model of types—which is the central motivation for the kinds in our system—derives from the indexed model proposed by Appel and McAllester [5]. In the indexed model, a type τ is a predicate on the tuple (k, v). It uses the judgment $v :_k \tau$ to say that a concrete value v has type τ to approximation k. This means that any program which runs for less than k instructions cannot tell whether v is of type τ. For example, if $v :_3$ *intlist* then v might be a pointer to a good list of integers, or to a list that has 3 or more good cons cells followed by a stray pointer; but not to a list that "goes wrong" in fewer than 3 dereferences.

A concrete value v is a tuple (s, x). In this tuple, s represents the current state of the machine, and x is an integer which represents the memory locations holding the value we are interested in. The state s is itself a tuple (a, m), where a identifies the allocated region of memory, and m is a map from memory locations to their contents.

In our calculus, a *closed* type is a predicate on (k, v), but a type expression with free variables must also have an environment ρ in which to look up de Bruijn indices. Thus a TML type is modeled as a predicate on (ρ, k, v), where ρ is a function from natural numbers to closed types.

A judgment $v :_k \tau$ makes sense if τ has no free type variables, and is an abbreviation for the semantic formula $\forall\rho.\ \tau(\rho, k, v)$.

2.1 Contractive and nonexpansive types

To reason about recursive types using the two-way subtyping $\mathsf{rec}\ \tau_F\ = \tau_F(\mathsf{rec}\ \tau_F)$, we must know that τ_F is *contractive* in its first argument. Intuitively, a type constructor F is contractive if it takes more machine instructions to reach the bottom of a value of type $F(\tau)$ than it does to reach the bottom of a value of type τ. Constructors such as box and $\rightarrow$ are contractive,[1] but constructors such as offset, $\cap$, and $\cup$ are not. Intuitively, a value of type $\tau_1 \cap \tau_2$ is no deeper than a value of type τ_1 or a value of type τ_2. However, these operators are all *nonexpansive*, meaning that the composition of offset or $\cap$ with a contractive operator is contractive.

Formally, we define contractiveness as follows. Let $\lfloor\phi\rfloor_k$ be a type representing the kth approximation to the type ϕ,

$$\lfloor\phi\rfloor_k = \lambda(j, v).\forall j < k.\, \phi(j, v)$$

and let the replacement of the ith binding in the type environment ρ by its jth approximation be given by $\rho[i \mapsto \lfloor\rho(i)\rfloor_j]$. Then we say a type function is f is contractive in its ith type argument iff

$$\forall\rho, k, v. \quad f^{k+1}(\rho, v) \equiv f^{k+1}(\rho[i \mapsto \lfloor\rho(i)\rfloor_k], v)$$

That is, in judging $v :_{k+1} f(\rho)$, we will never need to test $\rho(i)$ at approximation greater than k. This leads to a well-founded induction when we construct recursive types; indeed, Appel and McAllester call this *wellfoundedness* instead of contractiveness.

Similarly, we say that a type function is f is nonexpansive in its ith type argument iff

$$\forall\rho, k, v. \quad f^{k}(\rho, v) \equiv f^{k}(\rho[i \mapsto \lfloor\rho(i)\rfloor_k], v)$$

Appel and McAllester [5] show rules for deciding the well-formedness of compositions of single-argument type functions in the indexed model. For example, $\mathsf{rec}(F)$ is a well-formed type if F is contractive in its arguments. While techniques in MacQueen et al. [9], and Appel and McAllester [5] both can be extended to work for multiple argument functions, they are too unwieldy for writing machine-checkable implementations. Also, their rules are for very high-level operators. Since we deal with machine-level operators (which we use to build those high-level operators), we need different set of rules. One of the aims of our kinding system is to make this calculus easier for writing and proving lemmas, especially in an automated or semiautomated framework.

Consider, for example, the ML type

```
datatype 'a List = Cons of 'a * 'a List | Nil
```

A translation of this expression into high-level types would be $\Lambda\alpha.\mu\beta.((\alpha \times \beta) \cup nil)$ provided that $\alpha \times \beta$ and nil are disjoint.

[1]Following Appel and Felty, we build function closures in three stages. First-order continuations are modeled with the codeptr type constructor. A (higher-order) continuation is a closure, implemented as pair of codeptr and environment; and then the type of the environment is abstracted by existential quantification. Finally, functions are made by composing continuations in continuation-passing style.

Figure 1 shows the machine-level view of a *cons* and a *nil* integer instance of this type. Register r_1 contains zero denoting a *nil* cell. Register r_4 contains a nonzero value (314) denoting the *cons* cell. This value is interpreted as a pointer to a two-word structure. The first word contains integer 10, and the second contains a pointer to the next cell of the list.

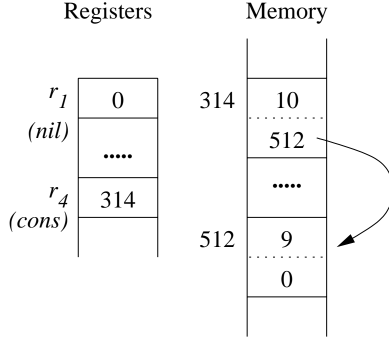

Figure 1: Representing *nil* and *cons* cells

Using low-level types, we describe the `List` type as

$$\mathsf{rec}\big((\mathsf{nonzero} \cap \mathsf{offset}\ 0\ (\mathsf{box}\ \underline{1}) \\ \cap\ \mathsf{offset}\ 1\ (\mathsf{box}\ \underline{0})) \\ \cup\ (\mathsf{const}\ 0)\big)$$

where any nonzero value has the nonzero type. Note that $\underline{0}$, the parameter to rec, appears deep inside the type expression within a box operator. The rec constructor requires its argument to be contractive in order to be well formed. Due to this wide separation of the binding of the type variable and its use in the type expression, we require a kinding system which can keep track of the contractive and nonexpansive properties of type variables.

2.2 Representable types

The inclusion of references in our semantic model is challenging. Ahmed, Appel, and Virga [2] give a semantic model of references. For this model to work, it is necessary for types to be *representable*. Informally, if a type is *representable* then a value with that type can be stored in a mutable cell in memory. We give a brief explanation of what it means for types to be representable.

In the Appel-Felty model of types [4], which can model immutable pointers and memory allocation but is too weak to model mutable references, a type is a predicate on memory M, the allocated set Λ, and a value v. M is a map from locations to values, while Λ is a finite set of "allocated" locations. But in conventional syntactic calculi with mutable references, Λ is a finite map from locations to types; that is, in addition to telling which locations are allocated, Λ tells what type of value we may store at each location. These syntactic calculi have a judgment $M : \Lambda$ saying that each element of the memory has an appropriate type.

A naive extension of the Appel-Felty model to a regime where Λ is a function from locations to types, yields the following problematic model:

Type	τ	:	$M \to \Lambda \to v \to o$.
Memory	M	:	$L \to v$.
Allocated Set	Λ	:	$L \to \tau$.
Values	v	:	$\{0, 1, ...\}$
Locations	L	:	$\{0, 1, ...\}$

The problem is that there is a circularity of definition in the semantic model: types are predicates on alloc-sets, and alloc-sets are predicates on types. Ahmed et al. remedy this situation by making the allocation set a map from location to *type syntax*, τ_{syn}. τ_{syn} is a tree (or Gödel number) which gives a syntactic description of the type. Although this removes the circularity, it requires another map from the type syntax to the semantic type. This map is the representation function repr. Given a type-syntax encoding, repr gives the corresponding semantic type. In this new setting we have:

Type syntax	τ_{syn}	:	$\{0, 1, 2, ...\}$
Allocated Set	Λ	:	$L \to \tau_{syn}$.
	repr	:	$\tau_{syn} \to \tau$.

For convenience we define a predicate "representable" such that

$$\text{representable} \stackrel{\text{def}}{=} \lambda\tau.\ \exists\tau_{syn}.\ \mathsf{repr}(\tau_{syn}) = \tau\ .$$

Types like $\mathsf{ref}(\tau)$ are meaningful in this model only if there is a syntax τ_{syn} for the type τ. Not all types are representable, though, and therefore this check is necessary. On the other hand, most type operators do not require that their argument-type be representable. Type expressions made only from operators like box and offset are representable and can go inside mutable references.

Quantifiers, however, are a bit more complex. To have meaningful quantified type expressions, we may choose to quantify over representable types, or over all types:

$$\begin{aligned} \forall_r &\stackrel{\text{def}}{=} \lambda F.\lambda(M, \Lambda, v).\forall\tau.\ \text{representable}(\tau)\ \Rightarrow (F\ \tau) \\ \forall_a &\stackrel{\text{def}}{=} \lambda F.\lambda(M, \Lambda, v).\forall\tau.\ (F\ \tau) \\ \exists_r &\stackrel{\text{def}}{=} \lambda F.\lambda(M, \Lambda, v).\exists\tau.\ \text{representable}(\tau)\ \wedge\ (F\ \tau) \\ \exists_a &\stackrel{\text{def}}{=} \lambda F.\lambda(M, \Lambda, v).\exists\tau.\ (F\ \tau) \end{aligned}$$

The operators $\forall_r$ and $\exists_r$ quantify over representable types, while $\forall_a$ and $\exists_a$ quantify over all types.

The semantic model of $\forall_r$ *must* refer to repr. Therefore, the repr relation cannot itself refer to $\forall_r$, and thus $\forall_r$ is unrepresentable. On the other hand, the model of $\forall_a$ does not refer to repr, and this means that repr can safely refer to $\forall_a$; that is, $\forall_a$ can be representable.

Over which types is it more useful to quantify? It turns out we need both. To represent the *abstype* feature of core ML, we need $\exists_r$, but to represent function closures we need $\exists_a$. To represent polymorphic functions we need $\forall_r$.

We need to keep track of representability of complex type expressions in an organized way. Representability is encoded into kinds in our system and the kinding system is then used externally to impose the representability requirement on the type expressions.

Throughout the 1980s, the interaction of references and polymorphism caused much trouble in the type theory of ML. Our semantic model, and the associated kinding system, provides a new explanation of this interaction.

3. TYPED MACHINE LANGUAGE

In our FPCC project, we use Typed Machine Language (TML) as an interface between the semantics built on higher-order logic and arithmetic, and the syntactic rules used to provide type checking (and safety proofs) for programs. That is, at the bottom we have higher-order logic (HOL), which is quite general and completely undecidable. At the top, we have a typed assembly language (TAL), which has a syntax-directed type-checking algorithm but is necessarily not very general: the TAL is suitable for a particular programming language as compiled by a particular compiler for a particular target machine. Our goal is to define the semantics of TAL in terms of HOL, and to prove all the typing rules of TAL as derived lemmas in HOL. We expect that these proofs will total about a hundred thousand lines of code, in the LF notation used by the Twelf [13] system.

A hundred thousand lines of software must be well modularized with appropriate abstraction layers, for it could never be built and maintained otherwise. Our most important abstraction layer between HOL and TAL is Typed Machine Language.

Because TAL must be syntax directed, it cannot include fully general intersection and union types, subtyping, quantification, and so on. TML has all the primitives necessary to construct the application-specific types used in TAL; therefore TML subtyping cannot be syntax-directed, or even decidable. (See Figure 2 for the full repertoire of TML type primitives.) We will prove the primitive TML subtyping rules in HOL, by hand, and check them by machine; then we will prove the TAL rules in the TML calculus, mostly by hand, and check them by machine. Finally we can apply the TAL rules to a given compiled program fully automatically.

Although TML is intended for the construction of TAL, and not to reason about programs directly, in this paper we will explain TML by applying it directly to machine-language programs.

For example, consider the ML program

```
datatype 'a List = Cons of 'a * 'a List | Nil
    ...
    case x : 'a List
      of Cons(h,t) => ...
       | Nil       => ...
```

As explained in section 2.1, we encode the datatype as

$$F \;=\; \mathsf{rec}\big((\mathsf{nonzero} \cap \mathsf{offset}\ 0\ (\mathsf{box}\ \underline{1}) \cap \mathsf{offset}\ 1\ (\mathsf{box}\ \underline{0})) \cup (\mathsf{const}\ 0)\big)$$

where the type expression on the right is an α list. The program can be compiled to machine code as

$\{r_1 : F(\mathsf{int})\}$

```
           if r1=0 goto NilCase
ConsCase:  r2 := mem[r1+0]
           ...
NilCase:   ...
```

The case discrimination implicit in the `if-goto` instruction requires that r_1 belong to a union type. Indeed, there is a union buried inside $F(\mathsf{int})$, but first we must beta-reduce (using the explicit substitution calculus) and then unfold the rec.

Kinds	κ	$::=$	$\Omega_0 \mid \Omega_R \mid \Omega_C \mid \Omega_{RC}$
Types	τ	$::=$	$\underline{n} \mid \top \mid \bot$
		$\mid$	$\mathsf{codeptr}(\tau) \mid \mathsf{offset}\ (\tau_1, \tau_2)$
		$\mid$	$\mathsf{box}(\tau) \mid \mathsf{ref}(\tau)$
		$\mid$	$\mathsf{rec}(\tau)$
		$\mid$	$\tau_1 \cap \tau_2 \mid \tau_1 \cup \tau_2$
		$\mid$	$\forall_r\ \tau \mid \exists_r\ \tau$
		$\mid$	$\forall_a\ \tau \mid \exists_a\ \tau$
		$\mid$	$\mathsf{const}(\tau)$
		$\mid$	$\mathsf{geq}(n) \mid \mathsf{leq}(n) \mid \mathsf{gt}(n) \mid \mathsf{lt}(n)$
		$\mid$	$\mathsf{plus}(\tau_1, \tau_2) \mid \mathsf{minus}(\tau_1, \tau_2)$
Naturals	n	$::=$	$\{0, 1, ...\}$

Figure 2: TML Kinds and Types

We have used our semantic model to prove all the conventional syntactic rules of the explicit substitution calculus, and we have implemented a logic program (in the Twelf system) that reduces expressions containing substitutions. Rendering $F(\mathsf{int})$ as $F[\mathsf{int} \cdot \mathsf{id}]$ and substituting gives us

$$G \;=\; \mathsf{rec}\big((\mathsf{nonzero} \cap \mathsf{offset}\ 0\ (\mathsf{box\ int}) \cap \mathsf{offset}\ 1\ (\mathsf{box}\ \underline{0})) \cup (\mathsf{const}\ 0)\big)$$

Now we can apply the UNFOLD rule,

$$\frac{v : F \qquad \text{“}F \text{ is contractive”}}{v : F(\mathsf{rec}\ F)}\ \textsc{unfold}$$

which results in the new typing judgment

$$r_1 :\ (\mathsf{nonzero} \cap \mathsf{offset}\ 0\ (\mathsf{box\ int}) \cap \mathsf{offset}\ 1\ (\mathsf{box}\ G)) \cup (\mathsf{const}\ 0)$$

After the conditional goto, where it is known that r_1 is not zero, we have:

$$r_1 : \mathsf{nonzero}\ \cap\ \big((\mathsf{nonzero} \cap \mathsf{offset}\ 0\ (\mathsf{box\ int}) \cap \mathsf{offset}\ 1\ (\mathsf{box}\ G)) \cup (\mathsf{const}\ 0)\big)$$

where $\mathsf{nonzero}$ is a type containing all the nonzero integers. Since $(\mathsf{nonzero}\ \cap\ \mathsf{const}\ 0) = \bot$, and using the distributivity law, we can prove

$$r_1 : (\mathsf{offset}\ 0\ (\mathsf{box\ int})\ \cap\ \mathsf{offset}\ 1\ (\mathsf{box}\ G))$$

and thus it is safe (at label `ConsCase`) to dereference r_1.

In FPCC, the UNFOLD rule is a theorem built upon the semantics of the rec type. It requires F to satisfy a contractiveness property, which we have written informally in the presentation above. Our kinding system, described in the next section, captures the semantics of contractiveness (and other properties) in a formal way, and hides the semantics under an abstraction layer, leaving only an easily manipulated syntax for the user of TML.

4. KINDING SYSTEM

Previous sections have motivated the need for a kinding system which is able to track the extensionality, representability, contractiveness, and constancy properties of TML types.

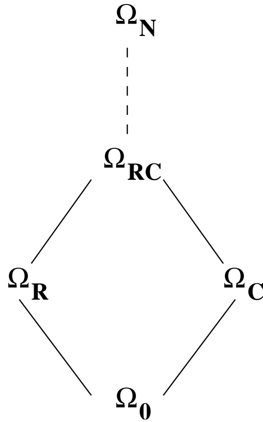

Figure 3: The TML kinding hierarchy as a lattice.

In this section we describe a hierarchy of kinds suited to that purpose, and give these kinds and the kinding judgment a semantic model that allows us to prove the kinding rules as lemmas. Last, we describe the kinding system's implementation as a logic program, and this program's use in assisting the users of TML in the writing of proofs.

4.1 Kinding hierarchy

The kinding system employs a hierarchy of five kinds: Ω_0, Ω_R, Ω_C, Ω_{RC}, and Ω_N. Except where noted, the reader may assume the presence of a weakening rule where one kind implies another.

- Ω_0 is the most inclusive kind, requiring only that its members respect basic properties such as extensionality over equivalent values and states.
- Ω_R implies Ω_0, and additionally requires its members to respect representability.
- Ω_C implies Ω_0, and additionally requires its members to respect contractiveness. While the kind hierarchy includes this element for completeness, in practice we have no need of it; it does not figure into the kinding judgments.
- Ω_{RC} implies both Ω_R and Ω_C.
- Ω_N is the kind of integer singleton types (those constructed from const). It can be proved that such types are both representable and contractive, that is, that Ω_N implies Ω_{RC}. However it is a matter of taste and style whether the TML calculus include the corresponding weakening rule. Here we demand that expressions having kind Ω_N be explicitly coerced to kind Ω_{RC}, in preference to the weakening rule.

The kinding hierarchy thus forms a lattice as per Figure 3. In the figure, the relation depicted is that of weakening. We show a dotted line from Ω_N to Ω_{RC} to denote that while such a weakening rule is possible, it remains a matter of taste whether to include it.

4.2 Semantics of the kinding judgment

In a conventional kinding system the syntax of the kinding judgment takes the following form:

$$x_0 :: \kappa_0, \ldots, x_n :: \kappa_n \vdash \tau :: \kappa$$

Because TML uses de Bruijn indices instead of variables, our kinding judgment need not mention the x_i explicitly. Rather, index $\underline{i}$ is assumed to have kind κ_i. We obtain therefore a judgment of the form

$$\kappa_0, \ldots, \kappa_n \vdash \tau :: \kappa$$

We call the array of kinds to the left of the turnstile Γ, and denote by $\Gamma(i)$ the ith kind in the array—the kind of the prospective term to be substituted for de Bruijn index $\underline{i}$.

4.2.1 Model I

Now we must assign semantics to the judgment $\Gamma \vdash \tau :: \kappa$. Let us summarize the concepts at hand and introduce a typographical convention. An open type τ may contain as subterms some number of free variables represented as de Bruijn indices. Such an open τ may be closed by applying it to an environment ρ that maps each index $\underline{i}$ to type $\rho(i)$. Note that $\rho(i)$ is itself necessarily a *closed* type. We use ϕ to denote closed types such as $\tau(\rho)$ and $\rho(i)$.

Since we associate a kind to each type variable, it is natural to think of modeling kinds as predicates on closed types. Since extensionality, representability, and constancy are properties of type expressions, this approach works well: Model I captures all the properties of interest except that of contractiveness.

Now in this model we render kinds as predicates on closed types ϕ. Since the kinding judgment $\Gamma \vdash \tau :: \kappa$ must ultimately involve the open type τ, and since const constructs an open type, some of the following definitions must make judicious use of a universally quantified ρ to close the open types. The kinds in this model are defined as follows:

$$\begin{aligned}
\Omega_0 &\stackrel{\text{def?}}{=} \lambda\phi.\ \text{extensional}(\phi) \\
\Omega_R &\stackrel{\text{def?}}{=} \lambda\phi.\ \text{representable}(\phi) \wedge \Omega_0(\phi) \\
\Omega_N &\stackrel{\text{def?}}{=} \lambda\phi.\ \exists n.\ \forall\rho.\ \phi = \mathsf{const}(n)(\rho)
\end{aligned}$$

and the the kinding judgment, thus:

$$\Gamma \vdash \tau :: \kappa \stackrel{\text{def?}}{=} \forall\rho.\ \Big(\forall i.\ \Gamma(i)\big(\rho(i)\big)\Big) \Rightarrow \kappa\big(\tau(\rho)\big)$$

Here, as before, ρ is an environment mapping indices to closed type expressions. Hence $\Gamma \vdash \tau :: \kappa$ says that τ belongs to κ under the assumption that each index $\underline{i}$ in τ has kind $\Gamma(i)$.

4.2.2 Model II

To the detriment of the previous model, contractiveness is not a property of types but rather of functions from types to types (hereafter "type functions"). Thus we must reformulate our kind calculus so that kinds are predicates on type functions. In such a formulation, it is simple to produce a definition for Ω_C:

$$\Omega_C \stackrel{\text{def}}{=} \lambda f.\ \text{contractive}(f) \wedge \Omega_0(f)$$

This formulation slightly complicates the definitions of Ω_0, Ω_R, and Ω_N. The solution is to require the type function under consideration to yield a result that is extensional, representable, or constant, respectively, regardless of its argument. Thus we write:

$$\Omega_0 \stackrel{\text{def}}{=} \lambda f.\ \forall \phi.\ \text{extensional}\big(f(\phi)\big)$$
$$\Omega_R \stackrel{\text{def}}{=} \lambda f.\ \forall \phi.\ \text{representable}\big(f(\phi)\big) \wedge \Omega_0(f)$$
$$\Omega_N \stackrel{\text{def}}{=} \lambda f.\ \exists n.\ \forall \phi.\ \forall \rho.\ f(\phi) = \mathsf{const}(n)(\rho)$$

For the revised formulation of the kinding judgment we retain the form of the previous incarnation. However, since the new regime raises the subjects of the kinding judgment from closed types to functions on closed types, we replace ρ, a map from integers to closed types $\underline{i}$, with R, a map from integers to functions on closed types. Similarly, on the right hand side of the implication arrow we replace $\tau(\rho)$ with the analogous type function constructed using R.

$$\Gamma \vdash \tau :: \kappa \stackrel{\text{def}}{=} \forall R.\ \Big(\forall i.\ \Gamma(i)\big(R(i)\big)\Big) \Rightarrow \kappa\Big(\lambda \phi.\ \tau\big(\lambda i.\ R(i)(\phi)\big)\Big)$$

We are left with the problem of how to relate each $R(i)$ to de Bruijn index $\rho(i)$. To solve this problem, consider the right hand side of the implication in the new kinding judgment. To relate type function $R(i)$ to $\underline{i}$, we can construct a ρ using R in its definition, such that $R(i)$ perturbs whatever would otherwise have been the image of i (here written as ϕ):

$$\rho_0 \stackrel{\text{def}}{=} \lambda i.\ R(i)(\phi)$$

Naturally, we wish this ρ_0 to be the environment used to close τ:

$$\phi_0 \stackrel{\text{def}}{=} \tau\big(\lambda i.\ R(i)(\phi)\big)$$

Now by λ-abstracting ϕ, we obtain a function on closed types suitable for offering to kind κ for judgment.

4.3 Kinding prover

We have implemented the inference rules for the kinding judgment (see Figure 4) as a logic program in the Twelf system. Each inference rule for the kinding judgment appears as a clause in the logic program. For example, consider the kinding rules for box and rec, which we write in notation as

$$\frac{\Gamma \vdash \tau :: \Omega_R}{\Gamma \vdash \mathsf{box}(\tau) :: \Omega_{RC}}\ , \qquad \frac{\Omega_R, \Gamma \vdash \tau :: \Omega_{RC}}{\Gamma \vdash \mathsf{rec}(\tau) :: \Omega_{RC}}\ .$$

In the logic program, these inference rules are realized as clauses thusly:

```
|-wellformed_box :
    |-wellformed Gamma (box @ Tau) omega_RC <-
    |-wellformed Gamma Tau omega_R .

|-wellformed_rec :
    |-wellformed Gamma (rec @ Tau) omega_RC <-
    |-wellformed (omega_R , Gamma) Tau omega_RC .
```

This Twelf presentation of the logic program's clauses differs from that of a traditional Prolog system's in two salient ways. First, note that each clause is labeled with the name of the corresponding inference rule. We will return to this point below and again in section 5. Second, for clauses Twelf uses the syntax

```
label :
    head <- subgoal_1 <- ... <- subgoal_n .
```

whereas a traditional Prolog system would use the syntax

```
    head :- subgoal_1 ,  ... ,  subgoal_n .
```

Twelf permits the use of higher-order abstract syntax, but we have no need of it here.

The user of the TML calculus would run the logic program by issuing to Twelf a directive of the form

```
tau = union
        @ (intersection
             @ nonzero
             @ (offset @ 0 @ (box @ (var @ 1)))
             @ (offset @ 1 @ (box @ (var @ 0))))
        @ (const @ 0) .

%solve witness :
    |-wellformed Gamma tau Kappa .
```

Ordinarily, `tau` is ground (the type expression of interest supplied by the user), and `Gamma` and `Kappa` are unification variables. Should the logic program succeed, Twelf instantiates `Gamma` and `Kappa`. Furthermore, Twelf binds to `witness` the witness to the program's success, a tree with nodes labeled with the names of clauses. The witness and the instantiated `Gamma` and `Kappa` may then be used in subsequent expressions of concern to the TML user. The user of the TML calculus is thus free to expend effort on interesting problems, leaving tedious syntax-directed judgments to the computer.

5. SEMANTIC PROOFS OF LOGIC PROGRAMS

Using the Twelf logical framework allows us the convenience to express our definitions, theorems and proof rules all in a uniform way. For example, the definition for the kind Ω_R is

```
omega_R :
    tm tml_kind =
  lam [f] omega_0 @ f and
    forall [tau] reppable @ repr @ tau imp
       reppable @ repr
          @ (lam [sigma] f @ (tau @ sigma)).
```

This statement in Twelf defines "`omega_R`" to be of Twelf metalogic type `tm tml_kind`, and having a definition following the "`=`" sign. Consider the kinding rule which states that the union of two types of kind Ω_{RC} is also of kind Ω_{RC}.

$$\frac{\tau_1 :: \Omega_{RC} \qquad \tau_2 :: \Omega_{RC}}{\tau_1 \cup \tau_2 :: \Omega_{RC}}\ \Omega_\cup_RC$$

Below is the corresponding semantic lemma in Twelf named "`omega_union_RC`". It is written in exactly the same syntax as the definition above:

```
omega_union_RC :
    pf (omega_RC @ T1) ->
    pf (omega_RC @ T2) ->
    pf (omega_0 @ (union @ T1 @ T2)) =
  [p1 : pf (omega_R @ T1)]
  [p2 : pf (omega_R @ T2)]
  ... p1 ...
  ... p2 ... .
```

The Twelf metalogic type of this lemma (the expression following the `:` sign) is

```
pf (omega_RC @ T1) ->
pf (omega_RC @ T2) ->
pf (omega_RC @ (union @ T1 @ T2)
```

and it tells us that given proofs of `T1` and `T1` having kind Ω_{RC}, we can derive a proof of `union @ T1 @ T2` having the kind Ω_{RC}. In an automated setting, we would require the equivalent of the Prolog rule:

```
omega_RC (union(T1, T2)) :-
    omega_RC (T1),
    omega_RC (T2).
```

This Prolog rule, however, is not backed by any semantic proofs. We can reuse the Twelf notation shown above to get an equivalent rule with a semantic proof:

```
RC_union_rule :
    pf (omega_RC @ (union @ T1 @ T2)) <-
    pf (omega_RC @ T1) <-
    pf (omega_RC @ T2) =
  [p1 : pf (omega_RC @ T1)]
  [p2 : pf (omega_RC @ T2)]
  (omega_union_RC @ p1 @ p2).
```

The last line gives us a justification for this "logic programming rule". As a result of this validity checking and uniformity of syntax, it is now possible for us to use Twelf to ensure the correctness of a logic program (composed of rules like `RC_union_rule`) which automatically generates proofs of well formedness of types.

$$\frac{\Gamma(i) = \kappa}{\Gamma \vdash \ \underline{i} :: \kappa}\ \text{WF_VAR}$$

$$\frac{}{\Gamma \vdash \ \top :: \Omega_{RC}}\ \text{WF_}\top \qquad \frac{}{\Gamma \vdash \ \bot :: \Omega_{RC}}\ \text{WF_}\bot$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_R}{\Gamma \vdash \ \mathsf{codeptr}\ \tau :: \Omega_{RC}}\ \text{WF_CPTR}$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_{RC} \quad \Gamma \vdash \ n :: \Omega_N}{\Gamma \vdash \ \mathsf{offset}\ (\tau_1, \tau_2) :: \Omega_{RC}}\ \text{WF_OFFSET}$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_R}{\Gamma \vdash \ \mathsf{box}\ \tau :: \Omega_{RC}}\ \text{WF_BOX} \qquad \frac{\Gamma \vdash \ \tau :: \Omega_R}{\Gamma \vdash \ \mathsf{ref}\ \tau :: \Omega_{RC}}\ \text{WF_REF}$$

$$\frac{\Omega_R, \Gamma \vdash \ \tau :: \Omega_{RC}}{\Gamma \vdash \ \mathsf{rec}\ \tau :: \Omega_{RC}}\ \text{WF_REC}$$

$$\frac{\Gamma \vdash \ \tau_1 :: \Omega_{RC} \quad \Gamma \vdash \ \tau_2 :: \Omega_{RC}}{\Gamma \vdash \ \tau_1 \cap \tau_2 :: \Omega_{RC}}\ \text{WF_}\cap$$

$$\frac{\Gamma \vdash \ \tau_1 :: \Omega_{RC} \quad \Gamma \vdash \ \tau_2 :: \Omega_{RC}}{\Gamma \vdash \ \tau_1 \cup \tau_2 :: \Omega_{RC}}\ \text{WF_}\cup$$

$$\frac{\Omega_{RC}, \Gamma \vdash \ \tau :: \Omega_{RC}}{\Gamma \vdash \ \forall_r\ \tau :: \Omega_0}\ \text{WF_}\forall_\text{r} \qquad \frac{\Omega_{RC}, \Gamma \vdash \ \tau :: \Omega_{RC}}{\Gamma \vdash \ \exists_r\ \tau :: \Omega_0}\ \text{WF_}\exists_\text{r}$$

$$\frac{\Omega_0, \Gamma \vdash \ \tau :: \Omega_0}{\Gamma \vdash \ \forall_a\ \tau :: \Omega_R}\ \text{WF_}\forall_\text{a} \qquad \frac{\Omega_0, \Gamma \vdash \ \tau :: \Omega_0}{\Gamma \vdash \ \exists_a\ \tau :: \Omega_R}\ \text{WF_}\exists_\text{a}$$

$$\frac{}{\Gamma \vdash \ \mathsf{const}(n) :: \Omega_N}\ \text{WF_CONST}$$

$$\frac{}{\Gamma \vdash \ \mathsf{geq}(n) :: \Omega_N}\ \text{WF_GEQ} \qquad \frac{}{\Gamma \vdash \ \mathsf{gt}(n) :: \Omega_N}\ \text{WF_GT}$$

$$\frac{}{\Gamma \vdash \ \mathsf{leq}(n) :: \Omega_N}\ \text{WF_LEQ} \qquad \frac{}{\Gamma \vdash \ \mathsf{lt}(n) :: \Omega_N}\ \text{WF_LT}$$

$$\frac{\Gamma \vdash \ \tau_1 :: \Omega_N \quad \Gamma \vdash \ \tau_2 :: \Omega_N}{\Gamma \vdash \ \mathsf{plus}(\tau_1, \tau_2) :: \Omega_N}\ \text{WF_+}$$

$$\frac{\Gamma \vdash \ \tau_1 :: \Omega_N \quad \Gamma \vdash \ \tau_2 :: \Omega_N}{\Gamma \vdash \ \mathsf{minus}(\tau_1, \tau_2) :: \Omega_N}\ \text{WF_}-$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_R}{\Gamma \vdash \ \tau :: \Omega_0}\ \text{WF_WEAK1} \qquad \frac{\Gamma \vdash \ \tau :: \Omega_C}{\Gamma \vdash \ \tau :: \Omega_0}\ \text{WF_WEAK2}$$

$$\frac{\Gamma \vdash \ \tau :: \Omega_{RC}}{\Gamma \vdash \ \tau :: \Omega_R}\ \text{WF_WEAK3} \qquad \frac{\Gamma \vdash \ \tau :: \Omega_{RC}}{\Gamma \vdash \ \tau :: \Omega_C}\ \text{WF_WEAK4}$$

Figure 4: Kinding (well-formedness) rules for TML

6. HIGHER-ORDER KINDS?

6.1 Object logic representation of kinds

We use the Twelf logical framework for our implementation of FPCC, which allows specification of type constructors in our object logic. Kinds such as Ω_R and Ω_N are represented by higher-order logic definitions such as `omega_R` and `omega_N`. Since these kinding predicates must be able to fit in the same slots in the kinding judgment, the (higher-order-logic) type of `omega_R` must be the same as the type of `omega_N`. Indeed, both predicates have the type

$$(\mathit{closedtype} \rightarrow \mathit{closedtype}) \rightarrow o\ .$$

It is natural to want to extend the kind system with higher-order kinds formed by a rule $\kappa ::= \kappa \rightarrow \kappa$. However, the kinding predicates representing higher-order kinds would have to have (higher-order-logic) types such as

$$(\mathit{closedtype} \rightarrow \mathit{closedtype}) \rightarrow (\mathit{closedtype} \rightarrow \mathit{closedtype}) \rightarrow o$$

and so on. These predicates would not fit into our representation framework in higher-order logic.

On the other hand, for any bounded degree of higher-order kinds, we could lift *all* of our kind predicates to a particular higher-order level. Indeed, to model the Featherweight Java type system using techniques described by League et al. [8], we will require first-level kind functions.

An alternate approach would be to permit different kind predicates to have different types in our underlying logical representation. Suppose we had N different types of kind predicates. Each kind of closed TML type would also be

represented by a differently-typed predicate in the logical representation. Then we could not use a single environment ρ to map type variables to closed types; we would need to parameterize τ by N different ρ variables. We could not use a single series of de Bruijn indices; we would need N different sets. We could not use just one *shift* operator in the explicit substitution calculus; we would need N^2 shift operators. We have decided to avoid this nightmare.

Because we have (at most) bounded higher-order kinds, it is not clear how our system could represent the proposed higher-order module system of ML [10]. In fact, we have not even worked out how to model the Standard ML module system [11]. We are compiling core ML to proof-carrying code [6], and building foundational safety proofs.

7. CONCLUSION

Our kind system for foundational PCC allows us to keep track of semantic properties of types; this is crucial to ensuring that derived types formed from compositions of basic types are sound. We believe that any type system for core ML *at the machine code level, that provides sufficient flexibility to optimizing compilers* must keep track of these properties.

While kinding rules in our system have machine-checked proofs, we also have a purely syntactic presentation of this system. This allows the complex semantic model to be hidden from users of this system.

We also have a logic program to automate proofs about well formedness of types using this kinding system. Using Twelf allows us to provide machine-checkable justification for these rules and thus prove the logic program correct.

8. REFERENCES

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 31–46. ACM Press, Jan 1990.

[2] Amal Ahmed, Andrew W. Appel, and Roberto Virga. Semantics of general references by a hierarchy of Gödel numberings. In *17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86, June 2002.

[3] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[4] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, New York, January 2000. ACM Press.

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, September 2001.

[6] Juan Chen, Dinghao Wu, Hai Fang, and Andrew W. Appel. Low-level typed assembly language. in preparation, 2001.

[7] N. G. deBruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–92, 1972.

[8] Christopher League, Valery Trifonov, and Zhong Shao. Type-preserving compilation of Featherweight Java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.

[9] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymophic types. *Information and Computation*, 71(1/2):95–130, 1986.

[10] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Proc. European Symposium on Programming (ESOP'94)*, pages 409–423, April 1994.

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

[12] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[13] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*, Berlin, July 1999. Springer-Verlag.