An Indexed Model of Recursive Types for Foundational Proof-Carrying Code

ANDREW W. APPEL
Princeton University
and
DAVID McALLESTER
AT&T Labs Research

The proofs of "traditional" proof carrying code (PCC) are type-specialized in the sense that they require axioms about a specific type system. In contrast, the proofs of foundational PCC explicitly define all required types and explicitly prove all the required properties of those types assuming only a fixed foundation of mathematics such as higher-order logic. Foundational PCC is both more flexible and more secure than type-specialized PCC.

For foundational PCC we need semantic models of type systems on von Neumann machines. Previous models have been either too weak (lacking general recursive types and first-class function-pointers), too complex (requiring machine-checkable proofs of large bodies of computability theory), or not obviously applicable to von Neumann machines. Our new model is strong, simple, and works either in λ-calculus or on Pentiums.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Theory

1. INTRODUCTION

Proof-carrying code (PCC) [Necula 1997] is a method of assuring that an untrusted program does no harm—does not access unauthorized resources, read private data, or overwrite valuable data. The provider of a PCC program must provide both the executable code and a machine-checkable proof that this code does not violate the safety policy of the host computer. The host computer does not run the given code until it has verified the given proof that the code is safe.

This research was supported in part by DARPA award F30602-99-1-0519 and by National Science Foundation grant CCR-9974553.
Authors' addresses: A. W. Appel, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544; email: appel@princeton.edu; D. McAllester, AT&T Labs Research, 180 Park Avenue, P.O. Box 971, Florham Park, NJ 07932-0971; email: dmac@research.att.com.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the cpoyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 0164-0925/01/1100–0657 $5.00

In most current approaches to PCC [Necula 1997; Morrisett et al. 1998b], the machine-checkable proofs are written in a logic with a built-in understanding of a particular type system. More formally, type constructors appear as primitives of the logic and certain lemmas about these type constructors are built into the verification system. The semantics of the type constructors and the validity of the lemmas concerning them are proved rigorously but without mechnical verification by the designers of the PCC verification system. We will call this type-specialized PCC.

Unlike type-specialized PCC, the foundational PCC described by Appel and Felty [2000] avoids any commitment to a particular type system. In foundational PCC the operational semantics of the machine code is defined in some logic $\mathcal{L}$, such as higher-order logic, that is suitably expressive to serve as a foundation of mathematics. $\mathcal{L}$ consists of a small set of axioms and definitional principles from which it is possible to build up most of modern mathematics. The operational semantics of machine instructions [Michael and Appel 2000] and safety policies [Appel and Felton 2001] are easily defined in higher-order logic. In foundational PCC the code provider must give both the executable code and a proof in $\mathcal{L}$ that the code satisfies the consumer's safety policy. In foundational PCC the proof must explicitly define, down to the foundations of mathematics, all required concepts, and must explicitly prove any needed properties of these concepts.

Foundational PCC has two main advantages over type-specialized PCC—it is more flexible and more secure. Foundational PCC is more flexible because the code producer can "explain" a novel type system or safety argument to the code consumer. It is more secure because its trusted base is smaller, consisting only of the foundational verification system, together with the definition of the machine instruction semantics and the safety policy. A verification system for higher-order logic can be made quite small [Harper et al. 1993; Pfenning 1994].

This paper presents a new type semantics intended to reduce the complexity of foundational type-theoretic proofs. The new semantics is particularly well suited for general recursive types, which are particularly tricky to handle semantically. Recursive types are closely related to domain equations, which were first given a semantics by Scott [Scott 1976; Schmidt 1986]. Recursive polymorphic types have been given a semantics in terms of metric spaces [MacQueen et al. 1986] and in terms of PER models of Turing machine computations [Mitchell and Viswanathan 1996]. We are also interested in polymorphism for our applications. The metric space approach is less powerful than the PER models—it models which terms are in which types, but does not properly model equivalences between terms—but it would be adequate for applications in proof-carrying code. A preliminary investigation by the first author, however, found no obvious definition of an appropriate metric for types on von Neumann machines. On the other hand the Mitchell-Viswanathan [1996] model might be adaptable to von Neumann machines, but would require years of effort "implementing" machine-checked proofs of basic results in computability theory.

Our new semantics is a term model,[1] with type judgements that are indexed: $v :_k \tau$ intuitively means that, in any computation running for no more than k steps, the value v behaves as if it were an element of the type τ. The recursive types of interest are well founded, in the sense that in order to determine whether $v :_k \tau$, it suffices to know whether $w :_j \tau$ for all values w and $j < k$. Well founded recursions always have unique fixed points.

Indexed types can also simplify the semantic treatment of the fixed point rule used to type recursive functions. This rule states that if $f : \alpha \to \alpha$ then $\mathrm{fix}(f) : \alpha$. The soundness of this rule is usually proved by defining a complete partial order (CPO) semantics and showing that all functions are monotone and continuous and hence have a least fixed point. Indexed types provide a direct soundness proof by induction on index, which avoids any use of semantic domains, term orders, or monotonicity.

Syntactic Versus Semantic Approaches

We are particularly interested in safety proofs based on type systems and in theorems stating that typability implies safety. Proofs that typability implies safety, are typically done by syntactic subject reduction—one proves that each step of computation preserves typability and that typable states are safe. However, in foundational PCC, the transmitted proof must contain all details down to the foundations of mathematics including the definitions of all concepts used. Foundational subject reduction theorems would require the explicit definition of inference rules and derivations in terms of foundational mathematical concepts—sets, pairs, and functions. They would also require case analyses over the different ways that a given type judgement might be derived. While this can all be done, we take a different approach to proving that typability implies safety.

We take a semantic approach, as pioneered in the NuPrl system [Constable et al. 1986] and as applied to proof-carrying code by Appel and Felty [2000]. In a semantic proof, one assigns a meaning (a semantic truth value) to type judgements. One then proves that if a type judgement is true, then the typed machine state is safe. One further proves that the type inference rules are sound, that is, if the premises are true, then the conclusion is true. This ensures that derivable type judgements are true, hence typable machine states are safe.

To contrast a semantic approach with syntactic subject reduction consider the following standard inference rule for typing applications.

$$\frac{\Gamma \vdash f : \alpha \to \beta, \quad \Gamma \vdash e : \alpha}{\Gamma \vdash (f\ e) : \beta}$$

A syntactic proof that typability implies safety must formalize the syntactic notion of typability. The above inference rule must be formalized as part of the definition of a relation $\vdash$ between syntactic type environments (mappings from

[1]There can be β-convertible terms v and w such that $v :_k \tau$ but not $w :_k \tau$ (and not $v :_{k'} \tau$ for sufficiently large k').

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

$$\frac{}{\pi_1\langle v_1,\ v_2\rangle \mapsto v_1} \qquad \frac{}{\pi_2\langle v_1,\ v_2\rangle \mapsto v_2} \qquad \frac{e_1 \mapsto e_1'}{\langle e_1,\ e_2\rangle \mapsto \langle e_1',\ e_2\rangle} \qquad \frac{e_2 \mapsto e_2'}{\langle v_1,\ e_2\rangle \mapsto \langle v_1,\ e_2'\rangle}$$

Fig. 1. Small step semantics.

variable to syntactic type expressions) and syntactic type judgements. This requires formalizing syntactic type expressions and formalizing the relation $\vdash$ as the least relation on syntactic expressions closed under a given set of inference rules.

The semantic approach avoids formalizing syntactic type expressions. Instead, one formalizes a type as a set of semantic values. One defines the operator $\rightarrow$ as a function taking two sets as arguments, and returning a set. The above type inference rule for application can then be replaced by the following semantic lemma in the foundational proof.

$$\frac{\Gamma \models f : \alpha \rightarrow \beta, \quad \Gamma \models e : \alpha}{\Gamma \models (f\ e) : \beta}$$

Although the two forms of the application type inference rule look very similar, they are actually significantly different. In the second rule, α and β range over semantic sets rather than type expressions. Furthermore, in the second version Γ is a function from program variables to semantic sets, rather than a function from program variables to type expressions. The relation $\models$ in the second version is defined directly in terms of a semantics for assertions of the form $e : \alpha$. The second "rule" is actually a lemma to be proved while the first rule is simply a part of the definition of the syntactic relation $\vdash$. For the purposes of foundational PCC, we view the semantic proofs as preferable to syntactic subject-reduction proofs because they lead to shorter and more manageable foundational proofs. The semantic approach avoids the need for any formalization of type expressions and avoids the formalization of proofs or derivations of type judgements involving type expressions.

2. INDEXED TYPES FOR THE LAMBDA CALCULUS

Before giving a semantic treatment of foundational PCC for von Neumann machine instructions, we give a semantic treatement of recursive types in the lambda calculus with cartesian products and the constant **0**. The syntax of lambda terms with products and **0** is defined by the following grammar.

$$e ::= x \mid \mathbf{0} \mid \langle e_1,\ e_2\rangle \mid \pi_1(e) \mid \pi_2(e) \mid \lambda x.e \mid (e_1\ e_2)$$

A term v is a *value* if it is **0**, a closed term of the form $\lambda x.e$, or a pair $\langle v_1, v_2\rangle$ of values. The small-step semantics (Figure 1) is entirely conventional. We write $e \mapsto^j e'$ to mean that there exists a chain of j steps of the form $e \mapsto e_1 \mapsto \cdots \mapsto e_j$ where e_j is e'. We write $e \mapsto^* e'$ if $e \mapsto^j e'$ for some $j \geq 0$. A term is *irreducible* if it has no successor in the step relation, that is irred(e) if e is a value or e is a "stuck" expression such as $\pi_1(\lambda x.e')$ or $0(e')$.

We say that e is safe for k steps, if for any reduction $e \mapsto^j e'$ of $j < k$ steps, either e' is a value or $e' \mapsto e''$. Note that any term is safe for 0 steps. A term e is called safe if it is safe for all $k \geq 0$. In this section we are interested in constructing methods for proving that a given term is safe. The semantic approach taken here, is based on types as sets rather than type expressions.

Definition 1. A *type* is a set τ of pairs of the form $\langle k, v\rangle$ where k is a nonnegative integer and v is a value, and where the set τ is such that if $\langle k, v\rangle \in \tau$ and $0 \leq j \leq k$ then $\langle j, v\rangle \in \tau$.

Informally, $\langle k, v\rangle \in \tau$ means that v "looks" like it belongs to type τ; perhaps v is not "really" a member of type τ, but any program of type $\tau \to \sigma$ must execute for at least k steps on v before getting to a stuck state.

Definition 2. For any closed expression e and type τ we write $e :_k \tau$ if whenever $e \mapsto^j v$ for $j < k$ and v irreducible, then $\langle k - j, v\rangle \in \tau$; that is,

$$e :_k \tau \equiv \forall j \forall e'. 0 \leq j < k \wedge e \mapsto^j e' \wedge \text{irred}(e') \Rightarrow \langle k - j, e'\rangle \in \tau$$

Intuitively, $e :_k \tau$ means that e behaves like an element of τ for k steps of computation. Note that if $e :_k \tau$ and $0 \leq j \leq k$ then $e :_j \tau$. Also, for a value v and $k > 0$, the statements $v :_k \tau$ and $\langle k, v\rangle \in \tau$ are equivalent. We now define various functions from sets to sets and an operation μ which takes a set functional F—a function from sets to sets—and returns a set that (we will show) is a fixed point of F. The μ operator allows us to define recursive types.

$$\begin{aligned}
\bot &\equiv \{\} \\
\top &\equiv \{\langle k, v\rangle \mid k \geq 0\} \\
\mathbf{int} &\equiv \{\langle k, \mathbf{0}\rangle \mid k \geq 0\} \\
\tau_1 \times \tau_2 &\equiv \{\langle k, (v_1, v_2)\rangle \mid \forall j < k.\ \langle j, v_1\rangle \in \tau_1 \wedge \langle j, v_2\rangle \in \tau_2\} \\
\sigma \to \tau &\equiv \{\langle k, \lambda x.e\rangle \mid \forall j < k\, \forall v.\ \langle j, v\rangle \in \sigma \Rightarrow e[v/x] :_j \tau\} \\
\mu F &\equiv \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\}
\end{aligned}$$

The definitions above can all be translated to higher-order logic. For example, our expression for μF would be written as,

$$\mu(F) = \lambda k \lambda v. \forall \tau. \text{ncomp}(F, k+1, \bot, \tau) \Rightarrow \tau\, k\, v$$

where $\text{ncomp}(F, k, g, h)$ means informally that $F^k(g) = h$ and can be defined in higher-order logic using a standard construction.[2]

Definition 3. A *type environment* is a mapping from lambda calculus variables to types. A *value environment* (also called a *ground substitution*) is a mapping from lambda calculus variables to values. For any type environment Γ and value environment σ we write $\sigma :_k \Gamma$ ("σ approximately obeys Γ") if for all variables $x \in \text{dom}(\Gamma)$ we have $\sigma(x) :_k \Gamma(x)$.

Finally, we define a semantic entailment relation $\models$. We write $\Gamma \models_k e : \alpha$ to mean that every free variable of e is mapped by Γ and

$$\forall \sigma. \sigma :_k \Gamma \Rightarrow \sigma(e) :_k \alpha$$

[2] $\text{ncomp}(f, n, x, y)$ can be defined as,

$$\forall g.\ (\forall z. g(f, 0, z, z)) \Rightarrow (\forall m, z_1, z_2. m > 0 \Rightarrow g(f, m-1, z_1, z_2) \Rightarrow g(f, m, z_1, f(z_2))) \Rightarrow g(f, n, x, y).$$

$$\Gamma \models x : \Gamma(x) \qquad \Gamma \models 0 : \text{int} \qquad \frac{\Gamma \models f : \alpha \to \beta \quad \Gamma \models e : \alpha}{\Gamma \models (f\ e) : \beta} \qquad \frac{\Gamma[x := \alpha] \models e : \beta}{\Gamma \models \lambda x.e : \alpha \to \beta} \qquad \frac{\Gamma \models e_1 : \alpha \quad \Gamma \models e_2 : \beta}{\Gamma \models \langle e_1, e_2 \rangle : \alpha \times \beta}$$

$$\frac{\Gamma \models e : \alpha \times \beta}{\Gamma \models \pi_1(e) : \alpha} \qquad \frac{\Gamma \models e : \alpha \times \beta}{\Gamma \models \pi_2(e) : \beta} \qquad \frac{\Gamma \models e : \mu F}{\Gamma \models e : F(\mu F)} \qquad \frac{\Gamma \models e : F(\mu F)}{\Gamma \models e : \mu F}$$

Fig. 2. Type inference lemmas.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x := \mu(\Lambda\alpha.\alpha \to \bot)] \models x : \mu(\Lambda\alpha.\alpha \to \bot)}{[x := \mu(\Lambda\alpha.\alpha \to \bot)] \models x : \mu(\Lambda\alpha.\alpha \to \bot),\quad [x := \mu(\Lambda\alpha.\alpha \to \bot)] \models x : (\mu(\Lambda\alpha.\alpha \to \bot)) \to \bot}}{[x := \mu(\Lambda\alpha.\alpha \to \bot)] \models (x\,x) : \bot}}{\models \lambda x.xx : (\mu(\Lambda\alpha.\alpha \to \bot)) \to \bot}}{\models \lambda x.xx : (\mu(\Lambda\alpha.\alpha \to \bot)) \to \bot \quad \models \lambda x.xx : \mu(\Lambda\alpha.\alpha \to \bot)}}{\models ((\lambda x.xx)\ (\lambda x.xx)) : \bot}$$

Fig. 3. A derivation of $\models \Omega : \bot$.

where $\sigma(e)$ is the result of replacing the free variables in e with their values under σ. We write $\Gamma \models e : \alpha$ if for all $k \geq 0$ we have $\Gamma \models_k e : \alpha$. We write $\models e : \alpha$ to mean $\Gamma_0 \models e : \alpha$ for the empty environment Γ_0.

Each of the type inference lemmas in Figure 2 states that if certain instances of the relation $\models$ hold, then certain other instances hold. Note that $\models$ can be viewed as a three place relation where $\Gamma \models e : \alpha$ means that the relation $\models$ holds on the type environment Γ, the term e, and the type α. Once we have proved the type inference lemmas in Figure 2, these lemmas can be used in the same manner as standard type inference rules to prove statements of the form $\Gamma \models e : \alpha$. We now observe that definitions 1, 2, and 3 immediately imply the following.

LEMMA 4. *If $\models e : \alpha$ then e is safe.*

Power of the type system. Our μ operator is powerful indeed. We can construct a typed **Y** combinator, but we will first explain the simpler term $\Omega = (\lambda x.xx)(\lambda x.xx)$, which reduces infinitely and therefore is safe. The derivation in Figure 3 shows how to use the type lemmas in Figure 2 to derive $\models \Omega : \bot$. By Lemma 4 we then have that Ω is safe. In the derivation $\Lambda\alpha.\alpha \to \bot$ is the set functional mapping the set α to the set $\alpha \to \bot$.

THEOREM 5. *The call-by-value fixed-point combinator $\mathbf{Y_v}$ is well typed*:

$$\models \lambda f.(\lambda x.f\,(\lambda z.xxz))(\lambda x.f\,(\lambda z.xxz)) : ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

for any α, β.

PROOF. Similar to the derivation in Figure 3, with the use (as necessary) of the hypotheses,

$$f : (\alpha \to \beta) \to (\alpha \to \beta)$$
$$x : \mu(\Lambda\delta.\delta \to (\alpha \to \beta)$$
$$z : \alpha$$

□

This means that our system can type recursive functions without requiring a primitive **fix** operator.

3. PROOFS OF THE TYPING LEMMAS

We now consider each of the type inference lemmas in Figure 2. Note that there is a type inference lemma for each case in the grammar of lambda terms plus two rules for the type constructor μ. The lemma for variables, stating that $\Gamma \models x : \Gamma(x)$, follows immediately from the definition of $\models$. The fact that int is a type, and the type inference lemma for 0 stating $\Gamma \models 0 : \text{int}$, both follow directly from the definition of int. We now consider the rules for applications and lambda expressions. First we have the following lemma which follows directly from the definition of $\rightarrow$.

LEMMA 6. *If α and β are types then $\alpha \rightarrow \beta$ is also a type.*

PROOF. By the definition of $\rightarrow$ it is obvious that $\alpha \rightarrow \beta$ is closed under decreasing index. □

We now prove the type theorems for application and lambda expressions.

LEMMA 7. *If e_1 and e_2 are closed terms and α and β are type sets such that $e_1 :_k \alpha \rightarrow \beta$ and $e_2 :_k \alpha$ then $(e_1 e_2) :_k \beta$.*

PROOF. Since $e_1 :_k \alpha \rightarrow \beta$ and $e_2 :_k \alpha$ we immediately have that both e_1 and e_2 are safe for k steps and that if e_1 generates a value in fewer than k steps, that value must be a lambda expression. Hence, the application $(e_1 e_2)$ either reduces for k steps without any top-level beta-reduction, or there exists a lambda expression $\lambda x.e$ and a value v such that $(e_1 e_2) \mapsto^j (\lambda x.e)\, v$ with $j < k$. In the first case we have that $(e_1 e_2)$ is safe for k steps and does not generate a value in less than k steps and hence $(e_1 e_2) :_k \beta$ (for any β). In the second case definitions 1 and 2 imply that $\langle k - j, \lambda x.e \rangle \in \alpha \rightarrow \beta$ and (using closure under decreasing index) $\langle k - j - 1, v \rangle \in \alpha$. The definition of $\rightarrow$ then implies $e[v/x] :_{k-j-1} \beta$. But we now have $(e_1 e_2) \mapsto^{j+1} e[v/x]$ and $e[v/x] :_{k-(j+1)} \beta$. These two statements imply $(e_1\, e_2) :_k \beta$. □

THEOREM 8 (APPLICATION). *If Γ is a type environment, e_1 and e_2 are (possibly open) terms, and α and β are types such that $\Gamma \models e_1 : \alpha \rightarrow \beta$ and $\Gamma \models e_2 : \alpha$ then $\Gamma \models (e_1\, e_2) : \beta$*

PROOF. We must prove that under the premises of the theorem and for any $k \geq 0$ we have $\Gamma \models_k (e_1 e_2) : \beta$. More specifically, for any σ such that $\sigma :_k \Gamma$ we must show $\sigma(e_1 e_2) :_k \beta$. By the premises of the theorem we have $\sigma(e_1) :_k \alpha \rightarrow \beta$ and $\sigma(e_2) :_k \alpha$. The result now follows from Lemma 7. □

THEOREM 9 (ABSTRACTION). *Let Γ be a type environment, let α and β be types, and let $\Gamma[x := \alpha]$ be the type environment that is identical to Γ except that it maps x to α. If $\Gamma[x := \alpha] \models e : \beta$ then $\Gamma \models \lambda x.e : \alpha \rightarrow \beta$.*

PROOF. As in Theorem 8, we must show that under the premises of the theorem we have that for any $k \geq 0$ and ground substitution σ such that $\sigma :_k \Gamma$ we have $\sigma(\lambda x.e) :_k \alpha \rightarrow \beta$. Suppose $\sigma :_k \Gamma$. Let v and $j < k$ be such that $v :_j \alpha$. By

the definition of $\rightarrow$ it now suffices to show that $\sigma(e[v/x]) :_j \beta$. Let $\sigma[x := v]$ be the ground substitution identical to σ except that it maps x to v. We now have that $\sigma[x := v] :_j \Gamma[x := \alpha]$. By the premise of the theorem, we then have that $\sigma[x := v](e) :_j \beta$; but this implies $\sigma(e[v/x]) :_j \beta$. □

LEMMA 10. *If α and β are types then so is $\alpha \times \beta$.*

LEMMA 11. *If α and β are types and e_1 and e_2 are closed terms such that $e_1 :_k \alpha$ and $e_2 :_k \beta$ then $\langle e_1, e_2\rangle :_k \alpha \times \beta$.*

PROOF. The proof is similar to the proof of Lemma 7. Again we have that e_1 and e_2 are safe for k steps. If $\langle e_1, e_2\rangle$ does not reduce to a pair of values within fewer than k steps, then we immediately have $\langle e_1, e_2\rangle :_k \alpha \times \beta$. So without loss of generality, we can assume that $\langle e_1, e_2\rangle \mapsto^j \langle v_1, v_2\rangle$ with $j < k$ and where v_1 and v_2 are values. Since $e_1 :_k \alpha$ and $e_2 :_k \beta$, we now have $v_1 :_{k-j} \alpha$ and $v_2 :_{k-j} \beta$, which implies $\langle v_1, v_2\rangle :_{k-j} \alpha \times \beta$. We now have $\langle e_1, e_2\rangle \mapsto^j \langle v_1, v_2\rangle$ and $\langle v_1, v_2\rangle :_{k-j} \alpha \times \beta$ and hence $\langle e_1, e_2\rangle :_k \alpha \times \beta$. □

The type inference theorem for pair expressions now follows from Lemma 3 in the same manner that Theorem 8 follows from Lemma 7.

LEMMA 12. *If α and β are types and e is a closed term such that $e :_k \alpha \times \beta$ then $\pi_1(e) :_k \alpha$ and $\pi_2(e) :_k \beta$.*

PROOF. We consider only the π_1 case. Since e is safe for k steps we can assume without loss of generality that $e \mapsto^j v$ for some value v and $j < k$. We now have $v :_{k-j} \alpha \times \beta$, which implies that v is a pair $\langle v_1, v_2\rangle$ with $v_1 :_{k-j-1} \alpha$; but we now have that $\pi_1(e) \mapsto^{j+1} v_1$ and $v_1 :_{k-(j+1)} \alpha$ and hence $\pi_1(e) :_k \alpha$. □

The type inference lemmas for projection terms follow from Lemma 3 in the same way that Theorem 8 follows from Lemma 7.

We have now proved all of the type inference lemmas except those for the type constructor μ. We will prove that the type inference for μ holds in the case where F is *well founded* and that all nontrivial type constructors built from type constants, $\rightarrow$, and $\times$ are well founded.

Definition 13. The *k-approximation* of a set is the subset of its elements whose index is less than k:

$$\text{approx}(k, \tau) = \{\langle j, v\rangle \mid j < k \wedge \langle j, v\rangle \in \tau\}$$

We have that if α is a type, then $\text{approx}(k, \alpha)$ is a type. We now define a notion of a well founded functional. Intuitively, a recursive definition of a type α is well founded if, in order to determine whether or not $e :_k \alpha$, it suffices to know $e' :_j \alpha$ for all terms e' and indices $j < k$.

Definition 14. A *well founded functional* is a function F from types to types such that for any type τ and $k \geq 0$ we have

$$\text{approx}(k+1, F(\tau)) = \text{approx}(k+1, F(\text{approx}(k, \tau)))$$

Note that if F is a function from types to types and α is a type then $F^k(\alpha)$ is a type for any $k \geq 0$.

LEMMA 15. *For F well founded and $j \le k$, for any τ, τ_1, τ_2,*

$$\text{approx}(j, F^j(\tau_1)) = \text{approx}(j, F^j(\tau_2)) \quad (1)$$
$$\text{approx}(j, F^j(\tau)) = \text{approx}(j, F^k(\tau)) \quad (2)$$

PROOF. (1) By induction.

$$\begin{aligned}
&\text{approx}(0, F^j(\tau_1)) = \bot = \text{approx}(0, F^j(\tau_2)).\\
&\text{approx}(j+1, F^{j+1}(\tau_1)) =\\
&\text{approx}(j+1, F(F^j(\tau_1))) =\\
&\text{approx}(j+1, F(\text{approx}(j, F^j(\tau_1)))) =\\
&\text{approx}(j+1, F(\text{approx}(j, F^j(\tau_2)))) =\\
&\text{approx}(j+1, F(F^j(\tau_2))) =\\
&\text{approx}(j+1, F^{j+1}(\tau_2))
\end{aligned}$$

(2) Using (1), taking $\tau_2 = F^{k-j}(\tau_1)$. □

This says that j applications of a well-founded functional to *any* type yields the same thing, to approximation j.

THEOREM 16. *If F is well founded, then μF is a type.*

PROOF. We must show that $\mu(F)$ is closed under decreasing index. Suppose that $\langle k, v\rangle \in \mu(F)$ and consider $j \le k$.

$$\begin{array}{ll}
\langle k, v\rangle \in \mu F & \\
\langle k, v\rangle \in F^{k+1}(\bot) & \text{by def'n of } \mu F\\
\langle j, v\rangle \in F^{k+1}(\bot) & \text{by def'n of type}\\
\langle j, v\rangle \in \text{approx}(j+1, F^{k+1}(\bot)) & \text{by def'n of approx}\\
\langle j, v\rangle \in \text{approx}(j+1, F^{j+1}(\bot)) & \text{by Lemma 15}\\
\langle j, v\rangle \in F^{j+1}(\bot) & \text{by def'n of approx}\\
\langle j, v\rangle \in \mu F & \text{by def'n of } \mu F
\end{array}$$
□

LEMMA 17. $\text{approx}(k, \text{approx}(k+1, \tau)) = \text{approx}(k, \tau)$. □

LEMMA 18. *If F is well founded,*

$$\text{approx}(k, \mu F) = \text{approx}(k, F^k \bot) \quad (a)$$
$$\text{approx}(k+1, F(\mu F)) = \text{approx}(k+1, F^{k+1} \bot) \quad (b)$$

PROOF. (a) For $k = 0$, each side is equivalent to $\bot$. For $k > 0$, each of the following lines is equivalent:

$$\begin{array}{ll}
\langle j, v\rangle \in \text{approx}(k, \mu F) & \\
j < k \wedge \langle j, v\rangle \in \mu F & \text{by def'n of approx}\\
j < k \wedge \langle j, v\rangle \in F^{j+1} \bot & \text{by def'n of } \mu F\\
j < k \wedge \langle j, v\rangle \in \text{approx}(j+1, F^{j+1} \bot) & \text{by def'n of approx}\\
j < k \wedge \langle j, v\rangle \in \text{approx}(j+1, F^k \bot) & \text{by Lemma 15}\\
j < k \wedge \langle j, v\rangle \in F^k \bot & \text{by def'n of approx}\\
\langle j, v\rangle \in \text{approx}(k, F^k \bot) & \text{by def'n of approx}
\end{array}$$

(*b*) Each of the following sets is equivalent.

$\text{approx}(k+1, F^{k+1}\bot)$	
$\text{approx}(k+1, F(F^k\bot))$	
$\text{approx}(k+1, F(\text{approx}(k, F^k\bot)))$	well-foundedness
$\text{approx}(k+1, F(\text{approx}(k, \mu F)))$	by (*a*)
$\text{approx}(k+1, F(\mu F))$	well-foundedness. □

LEMMA 19. *If F is well founded*, $\text{approx}(k, \mu F) = \text{approx}(k, F(\mu F))$.

PROOF. Each of the following sets is equivalent

$\text{approx}(k, \mu F)$	
$\text{approx}(k, F^k\bot)$	by Lemma 18a
$\text{approx}(k, F^{k+1}\bot)$	by Lemma 15
$\text{approx}(k, \text{approx}(k+1, F^{k+1}\bot))$	by Lemma 17
$\text{approx}(k, \text{approx}(k+1, F(\mu F)))$	by Lemma 18b
$\text{approx}(k, F(\mu F))$	by Lemma 17. □

THEOREM 20. *If F is well founded, then $\mu F = F(\mu F)$. Hence the type inference lemmas for μ in Figure 2 hold for any well founded functional F.*

PROOF. We have that $\langle k, v\rangle \in \mu F$ iff $\langle k, v\rangle \in \text{approx}(k+1, \mu F)$ iff $\langle k, v\rangle \in \text{approx}(k+1, F(\mu F))$ iff $\langle k, v\rangle \in F(\mu F)$. □

Theorem 20 justifies the derivation in Figure 3 provided that one can show that the functional $\Lambda\alpha.\alpha \to \bot$ is well founded. Let α be the set $\mu(\Lambda\alpha.\alpha \to \bot)$. Intuitively, α is defined by the recursive definition $\alpha = \alpha \to \bot$. We can think of the type α as a predicate on pairs $\langle k, v\rangle$. A recursive definition of a predicate is well founded if every recursive call is on a smaller argument. For the recursive definition $\alpha = \alpha \to \bot$ we have that $\langle k, \lambda x.e\rangle \in \alpha$ iff for all $j < k$ and $\langle j, v\rangle \in \alpha$ we have $e[v/x] :_j \bot$. So we have an infinite number of recursive calls to α, each of the form $\langle j, v\rangle \in \alpha$ with $j < k$—all recursive calls are on simper arguments. So we get that the recursive definition $\alpha = \alpha \to \bot$ is well founded. A more rigorous proof that $\Lambda\alpha.\alpha \to \bot$ is well founded can be given using Lemma 22 below.

In the remainder of this section we use *type constructor* to mean a function from type to type.

Definition 21. A nonexpansive type constructor F is one such that

$$\text{approx}(k, F(\tau)) = \text{approx}(k, F(\text{approx}(k, \tau)))$$

The constructor $\Lambda\alpha.\alpha$ is nonexpansive but not well founded. Other examples (definable as extensions to the tiny type system of this paper) are $\Lambda\alpha.\alpha \cap \tau$, $\Lambda\alpha.\alpha \cup \tau$, and the `offset` constructor of Appel and Felty [2000].

LEMMA 22 (NONEXPANSIVE CONSTRUCTORS).

a. Every well founded constructor is nonexpansive.

b. $\Lambda\alpha.\alpha$ is nonexpansive.

c. $\Lambda\alpha.\tau$, where α is not free in τ, is well founded.

d. The composition of nonexpansive constructors is nonexpansive.

e. The composition of a nonexpansive constructor with a well founded constructor (in either order) is well founded.

f. If F and G are nonexpansive, then $\Lambda\alpha.F\alpha \to G\alpha$ is well founded.

g. If F and G are nonexpansive, then $\Lambda\alpha.F\alpha \times G\alpha$ is well founded.

PROOF. In the following we assume that F and G are nonexpansive and that H is well founded.

a. $\text{approx}(0, H(\alpha)) = \text{approx}(0, H(\text{approx}(0, \alpha)))$;

$$\begin{array}{ll} \text{approx}(k+1, H(\alpha)) = & \\ \text{approx}(k+1, H(\text{approx}(k, \alpha))) = & \text{by Definition 14} \\ \text{approx}(k+1, H(\text{approx}(k, \text{approx}(k+1, \alpha)))) = & \text{by Lemma 17} \\ \text{approx}(k+1, H(\text{approx}(k+1, \alpha)))) & \text{by Definition 14.} \end{array}$$

b. Let I be $\Lambda\alpha.\alpha$. Then $\text{approx}(k, I(\alpha)) = \text{approx}(k, I(\text{approx}(k, \alpha)))$.

c. Let K be a constant function. Then $\text{approx}(k+1, K(\alpha)) = \text{approx}(k+1, K(\text{approx}(k, \alpha)))$.

d. $\text{approx}(k, F(G(\alpha))) = \text{approx}(k, F(\text{approx}(k, G(\text{approx}(k, \alpha))))) = \text{approx}(k, F(G(\text{approx}(k, \alpha))))$ by Definition 21 (twice).

e. $\text{approx}(k+1, F(H(\alpha))) = \text{approx}(k+1, F(\text{approx}(k+1, H(\text{approx}(k, \alpha))))) = \text{approx}(k+1, F(H(\text{approx}(k, \alpha))))$.
$\text{approx}(k+1, H(F(\alpha))) = \text{approx}(k+1, H(\text{approx}(k, F(\text{approx}(k, \alpha))))) = \text{approx}(k+1, H(F(\text{approx}(k, \alpha))))$.

f. By the definition of $\to$ we have $\text{approx}(k+1, \alpha \to \beta) = \text{approx}(k+1, \text{approx}(k, \alpha)) \to \text{approx}(k, \beta))$.
This gives the following.

$$\begin{array}{l} \text{approx}(k+1, F(\alpha) \to G(\alpha)) = \\ \text{approx}(k+1, \text{approx}(k, F(\alpha)) \to \text{approx}(k, G(\alpha))) = \\ \text{approx}(k+1, \text{approx}(k, F(\text{approx}(k, \alpha))) \to \text{approx}(k, G(\text{approx}(k, \alpha)))) = \\ \text{approx}(k+1, F(\text{approx}(k, \alpha)) \to G(\text{approx}(k, \alpha))) \end{array}$$

by the above, then Definition 21, and again by the above.

g. By the definition of $\times$ we have $\text{approx}(k+1, \alpha \times \beta) = \text{approx}(k+1, \text{approx}(k, \alpha)) \times \text{approx}(k, \beta))$.
This gives the following.

$$\begin{array}{l} \text{approx}(k+1, F(\alpha) \times G(\alpha)) = \\ \text{approx}(k+1, \text{approx}(k, F(\alpha)) \times \text{approx}(k, G(\alpha))) = \\ \text{approx}(k+1, \text{approx}(k, F(\text{approx}(k, \alpha))) \times \text{approx}(k, G(\text{approx}(k, \alpha)))) = \\ \text{approx}(k+1, F(\text{approx}(k, \alpha)) \times G(\text{approx}(k, \alpha))) \end{array}$$

□

LEMMA 23. *If F is the identity constructor $\Lambda\alpha.\alpha$, then $\mu F = F(\mu F)$.*

PROOF. $F^j(\bot) = \bot$, so both sides are equal to $\bot$. □

Quantified types. We can also model existential types—useful for data abstraction, and universal types—useful for polymorphic functions. The semantic

constructors are,

$$\exists F \equiv \bigcup_{\tau \in \text{type}} F\tau \qquad \forall F \equiv \bigcap_{\tau \in \text{type}} F\tau$$

where $\tau \in \text{type}$ means, as usual, that τ is closed under decreasing index.

THEOREM 24 (TYPING RULES FOR QUANTIFIED TYPES).

$$\frac{\forall \tau \in \text{type}.\ F\tau \in \text{type}}{(\exists F) \in \text{type} \quad (\forall F) \in \text{type}} \tag{a}$$

$$\frac{\tau \in \text{type} \quad \Gamma \models v : F\tau}{\Gamma \models v : \exists F} \tag{b}$$

$$\frac{\Gamma \models v : \exists F}{\exists \tau \in \text{type}.\ \Gamma \models v : F\tau} \tag{c}$$

$$\frac{\forall \tau \in \text{type}.\ \Gamma \models v : F\tau}{\Gamma \models v : \forall F} \tag{d}$$

$$\frac{\Gamma \models v : \forall F}{\forall \tau \in \text{type}.\ \Gamma \models v : F\tau} \tag{e}$$

These rules all follow trivially from the definitions. However, rules *b*–*e* are rather operational; they don't look exactly like the usual type-checking rules for quantified types, which involve the explicit management of a set of type variables. It should be possible to define an extended notion of semantic entailment $\Delta, \Gamma \models_k e : \tau$, where Δ is a set of type variables, to support this form of type checking.

Even with our current definitions we can state theorems such as parametricity. For example, we can prove that the only functions of type $\forall\alpha.\alpha \to \alpha$ are the empty (always nonterminating) function and the identity function; the usual method of considering, for each value v, the singleton type τ_v works straightforwardly in our semantics.

THEOREM 25. *The typing rules of Figure* 2 *are sound.*

PROOF. $\Gamma \models x : \Gamma(x)$ is directly from the definitions of $\models$ and $e : \tau.\ \Gamma \models 0 : \text{int}$ is similarly trivial. The rules for application have been proved as Theorems 8 and 9; pairing and projection are Lemmas 3 and 12.

By Lemma 22, any type constructor $\Lambda\alpha.\tau$, where τ is built from α and the operators **int**, $\top$, $\bot$, $\times$, $\to$ is either well founded or the identity. Thus, by Theorem 20 and Lemma 23, $\mu F = F(\mu F)$. □

By Lemma 4, any well typed closed expression is safe. Therefore we have a model of general recursive types that is powerful enough to prove safety of any simply typed λ-expressions. Theorem 24 hints at how to generalize this to calculi with existential and polymorphic types.

4. AN INDEXED PER MODEL

We have shown a model of types in which we can reason about the membership of terms in types. Even more useful, is a model in which we can reason about the

equivalence of terms. This allows us to use the model to prove, for example, that a compiler optimization has correctly transformed an expression. Just as useful, is the ability to prove that some function f produces the same (i.e., equivalent) result, independent of the representation of its argument; this permits more perfect information hiding across interfaces.

Readers not interested in PER's (partial equivalence relations) can skip this section, as later sections do not depend on it. Moreover, the fact that our semantics doesn't require operational equivalence, is a significant advantage in some situations. For example, defining operational equivalence for a calculus with mutable references requires characterization of the set of references visible to a given expression—a type-and-effect discipline—but when all we need is proofs of safety, we can avoid all that work by using the simple non-PER model described in the previous section.

Our indexed model extends easily to PER models of types. We define a type as a set of triples $\langle k, v, w\rangle$, with the (informal) meaning that in any computation of no more than k steps, v approximates w—that is, if $f(v)$ halts in k steps, then $f(w)$ also halts and yields the same result.

We extend this relation from values to expressions, using the four-place relation $e \le f :_k \tau$, defined as

$$\begin{aligned} e \le f :_k \tau \equiv \forall j \forall e'. 0 < j < k \wedge e \mapsto^j e' \wedge \mathrm{irred}(e') \\ \Rightarrow \exists f'. f \mapsto^* f' \wedge \langle k - j, e', f'\rangle \in \tau \end{aligned}$$

The statement $e :_k \tau$ is an abbreviation for $e \le e :_k \tau$, and serves as a "conventional" typing judgement.

$$\begin{aligned} \bot &\equiv \{\} \\ \mathbf{int} &\equiv \{\langle k, \mathbf{0}, \mathbf{0}\rangle \mid k \ge 0\} \\ \tau_1 \times \tau_2 &\equiv \{\langle k, (v_1, v_2), (w_1, w_2)\rangle \mid \forall j < k.\ \langle j, v_1, w_1\rangle \in \tau_1 \wedge \langle j, v_2, w_2\rangle \in \tau_2\} \\ \sigma \to \tau &\equiv \{\langle k, \lambda x.e, \lambda y.f\rangle \mid \forall j < k\ \forall v, w.\ \langle j, v, w\rangle \in \sigma \Rightarrow e[v/x] \le f[w/y] :_j \tau\} \\ \mu F &\equiv \{\langle k, v, w\rangle \mid \langle k, v, w\rangle \in F^{k+1}(\bot)\} \end{aligned}$$

As before, we define well-typed substitutions, and we define typing entailments $\Gamma \models e \le f : \tau$.

$$\begin{aligned} \sigma_1 \le \sigma_2 :_k \Gamma &\equiv \mathrm{dom}\ \sigma_1 = \mathrm{dom}\ \sigma_2 = \mathrm{dom}\ \Gamma \wedge \forall x.\ \sigma_1(x) \le \sigma_2(x) :_k \Gamma(x) \\ \Gamma \models_k e \le f : \tau &\equiv \forall \sigma_1, \sigma_2.\ \sigma_1 \le \sigma_2 :_k \Gamma \Rightarrow \sigma_1(e) \le \sigma 2(f) :_k \tau \\ \Gamma \models e \le f : \tau &\equiv \forall k.\ \Gamma \models_k e \le f : \tau \end{aligned}$$

Now we can prove the type entailment theorems corresponding to Figure 2.

LEMMA 26. *If e_1, f_1, e_2, f_2 are closed terms, and α, β are types such that $e_1 \le f_1 :_k \alpha \to \beta$ and $e_2 \le f_2 :_k \alpha$ then $(e_1\, f_1) \le (e_2\, f_2) :_k \beta$.*

PROOF. By analogy with the proof of Lemma 7. Both e_1 and e_2 are safe for k steps. If $e_1 \mapsto^{j_1} v_1$ with $j_1 < k$, then v_1 must be a lambda expression $\lambda x.e$ and $f_1 \mapsto^* f_1'$ with $\langle k - j_1, \lambda x.e, f_1'\rangle \in \alpha \to \beta$. Hence, the application $e_1 e_2$ either reduces for k steps without any top-level beta-reduction—in which case $e_1 e_2 \le f :_k \tau$ for any f and τ—or $(e_1 e_2) \mapsto^{j_1} (\lambda x.e) e_2 \mapsto^{j_2} (\lambda x.e) v$ with $j_1 + j_2 < k$, $f_2 \mapsto^* f_2'$, and $\langle k - j_2, v, f_2'\rangle \in \alpha$.

Since the only values in $\alpha \to \beta$ are lambdas, $f_1' = \lambda y.f$ for some y and f. By decreasing index, $\langle k - j_1 - j_2 - 1, v, f_2' \rangle \in \alpha$, and by the definition of $\alpha \to \beta$ we have $e[v/x] \leq f[f_2/y] :_{k-j_1-j_2-1}$. Either $e[v/x]$ steps for another $k - j_1 - j_2 - 1$—in which case e_1e_2 has now stepped for k steps and $e_1e_2 \leq f :_k \tau$ for any f and τ—or (because it is approximately well typed) reduces to a value v_3 in j_3 steps, with $j_3 < k - j_1 - j_2 - 1$. Then $f[f_2/y] \mapsto^* f_3$ and $\langle k - j_1 - j_2 - 1 - j_3, v_3, f_3 \rangle \in \beta$. Thus, $e_1e_2 \mapsto^{j_1+j_2+1+j_3} v_3$; but $f_1 f_2 \mapsto^* f_3$, with the required relation between v_3 and f_3. □

THEOREM 27 (APPLICATION).

$$\frac{\Gamma \models e_1 \leq f_1 : \alpha \to \beta \quad \Gamma \models e_2 \leq f_2 : \alpha}{\Gamma \models (e_1e_2) \leq (f_1 f_2) : \beta}$$

PROOF. By analogy with Theorem 8, but using Lemma 26. □

COROLLARY 28.

$$\frac{\Gamma \models e_1 : \alpha \to \beta \quad \Gamma \models e_2 : \alpha}{\Gamma \models (e_1e_2) : \beta}$$

THEOREM 29 (ABSTRACTION).

$$\frac{\Gamma[x := \alpha] \models e \leq f : \beta}{\Gamma \models (\lambda x.e) \leq (\lambda x.f) : \alpha \to \beta}$$

PROOF. We must show that for any k and σ_1, σ_2 such that $\sigma_1 \leq \sigma_2 :_k \Gamma$, we have $\sigma_1(\lambda x.e) \leq \sigma_2(\lambda x.f) :_k \alpha \to \beta$. Let v, w and $j < k$ be such that $v \leq w :_j \alpha$. By the definition of $\to$ it suffices to show that $\sigma_1(e[v/x]) \leq \sigma_2(f[w/x]) :_j \beta$. We can extend σ_1 and σ_2 so that $\sigma_1[x := v] \leq \sigma_2[x := w] :_j \Gamma[x := \alpha]$. By the premise of the theorem we have $\sigma_1[x := v] \leq \sigma_2[x := w] :_j \beta$. This implies $\sigma_1(e[v/x]) \leq \sigma_2(f[w/x]) :_j \beta$. □

COROLLARY 30.

$$\frac{\Gamma[x := \alpha] \models e : \beta}{\Gamma \models \lambda x.e : \alpha \to \beta}$$

LEMMA 31. *If α and β are types (i.e., closed under decreasing index), then so are $\bot$, **int**, $\alpha \times \beta$, and $\alpha \to \beta$.*

Definitions, Lemmas, and Theorems 13–20 hold, using sets of triples instead of sets of pairs. That is, the definition of approx(k, τ) and well-foundedness, and the lemmas and theorems about well founded type constructors, up to and including $\mu F = F(\mu F)$, are written in exactly the same way.

LEMMA 32. *All the statements (a)–(g) of Lemma 22, and Lemma 23, hold for indexed-per type constructors.*

THEOREM 33. *Any type constructor F expressible in the "syntax" of constructors **int**, $\times$, $\to$, μ is well founded, so therefore $\mu F = F(\mu F)$.*

LEMMA 34. *If $\models e : \alpha$ then e is safe.*

In the PER model, however, we get more than just the lemma that typability implies safety. We also get congruence and extensionality results: a well-typed

function must map equivalent arguments to equivalent results, and if two functions behave the same, then the type system judges them equivalent.

Define $e \sim f : \tau$ to mean $e \leq f : \tau \wedge f \leq e : \tau$.

THEOREM 35 (CONGRUENCE).

$$\frac{\Gamma \models e_1 \sim f_1 : \alpha \to \beta \quad \Gamma \models e_2 \sim f_2 : \alpha}{\Gamma \models (e_1 e_2) \sim (f_1 f_2) : \beta}$$

PROOF. By Theorem 27. □

THEOREM 36 (EXTENSIONALITY).

$$\frac{\forall v, w.\ \models v \sim w : \alpha \Rightarrow \models f v \sim g w : \beta}{\models f \sim g : \alpha \to \beta}$$

PROOF. From the definition of $\alpha \to \beta$. □

CLAIM (OBSERVATIONAL EQUIVALENCE). If $e \sim f : \tau$ then e and f have the same observable behavior in any context of type τ.

PROOF. By the definition of $\sim$, via Theorem 27 (and a similar theorem for pairing) e is *applicatively equivalent* to f. Observational equivalence should follow via an adaption (for this calculus) of Milner's Context Lemma [Milner 1977]. □

5. PROOF-CARRYING CODE

A fully mechanized foundational safety proof will have two parts:

(1) Machine-checked proof of the typing rules;
(2) Mechanized application of the typing rules.

This paper concentrates on (1). Section 2 has given proofs of typing rules for lambda-calculus in excruciating detail, and yet that section is still fairly short. This should be good evidence for mechanizability. We will not repeat all these proofs in the same level of detail for von Neumann machines, but the proofs are almost as simple.

Mechanized application of the typing rules is another story entirely. The strategy successfully used by Necula [1998] is to use Nelson-Oppen collaborating decision procedures. We intend to follow a different strategy, using our indexed model of types to directly model Typed Assembly Language [Morrisett et al. 1998b]. However, both of these are beyond the scope of this paper. We will show informal proof techniques, but not decision procecures.

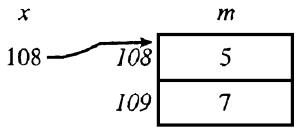
For the application of proof-carrying code, we need a soundness proof of recursive types, not in lambda-calculus, but on a von Neumann machine—in Pentium instructions, for example. The step relation of interest is not a predicate on pairs of expressions $e_1 \mapsto e_2$, but on pairs of machine states $(r_1, m_1) \mapsto (r_2, m_2)$, where r is the contents of the register bank and m is the contents of the memory—the execution of one instruction can take the machine from state (r_1, m_1) to state (r_2, m_2) [Appel and Felty 2000; Michael and Appel 2000].

On such machines it is most convenient to define simpler type primitives than the cartesian product and function arrow of lambda calculus:

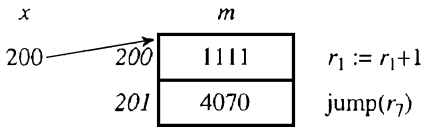
int. The type of one-word machine integers.
const(n). The singleton type containing only the integer value n.
ref(τ). Pointer to a memory location containing a value of type τ.
offset(n, τ). A value that, if you add n to it, yields a value of type τ.
$\sigma \cap \tau$. The intersection of σ and τ. The (boxed) cartesian product $\sigma \times \tau$ can be built from **offset**(0, **ref**(σ))$\cap$**offset**(1, **ref**(τ)); a record with a σ value in the first field and a τ value in the second field.
$\sigma \cup \tau$. The union of σ and τ. A (tagged) disjoint union $\sigma + \tau$ can be built from (**const**(0) $\times \sigma$) $\cup$ (**const**(1) $\times \tau$), that is, a record with a tag in the first field and (depending on the tag value) either a σ or a τ in the second field.
$\exists\alpha.\tau$. An existential type.
codeptr(τ). A first-order continuation; that is, an address in the machine code that is safe to jump to as long as an argument of type τ is passed in a designated register. Higher-order continuations (i.e., closures) can be constructed using first-order closures and existential types; higher-order functions can be constructed from higher-order closures [Minamide et al. 1996].

A *value* is a pair (m, x) where m is a finite partial function from integers to integers (a partial memory) and x is an integer (typically representing an address).[3]

To represent a pointer data structure that occupies a certain portion of the machine's memory, we let x be the root address of that structure, and the domain of m, the set of addresses occupied by the data. For example, the boxed pair of integers $\langle 5, 7\rangle$ represented at address 108 would be represented as the value $(\{108 \mapsto 5, 109 \mapsto 7\}, 108)$.



To represent a function (actually, a continuation) value, we let x be the entry address of the function, and the domain of m be the set of addresses containing machine instructions of the function. Here is the function $f(x, k) = k(x + 1)$, assuming that x is in register 1, and k is passed in register 7:



We assume that one of the registers is the program counter—for example, register $r(37)$ could be the program counter, pc = 37. Then a machine

[3]Appel and Felty use a triple (a, m, x) where m is a total function and a is the set describing the domain of interest—the two formulations are equivalent.

state (r, m) in which we have just jumped to location 200 has the property $r(\text{pc}) = 200$.

The step relation $(r, m) \mapsto (r', m')$ is defined on total functions m and m'; that is, a machine instruction might fetch from any location. Any particular data structure (i.e., value (m_1, x_1)) occupies only a finite portion of memory (the domain of m_1 is finite). In order for the program to create and initialize new data structures, it must know what addresses in m are not part of any already existing data structures. That is, at any time all existing values live in *allocated* address of the heap, and unallocated addresses can be used for new data structures, and the allocated set must be computable from the current contents of the register bank and memory. We model this with a function $\text{alloc}(r, m)$ that takes a register bank and memory and returns a set of addresses (integers). An example of a simple alloc function is

$$\text{alloc}(r, m) = \{x \mid 0 \leq x < r(6)\}$$

where register 6 points to the boundary between allocated and unallocated locations. To allocate and initialize a new data structure, the program would store at locations $r(6), r(6)+1, \ldots$ and then increment $r(6)$.

The machine has a step relation $\mapsto$ that models the decoding and execution of one instruction. We have described how to model this relation in previous work [Appel and Felty 2000; Michael and Appel 2000] and will not repeat it here. The important property of our axiomatization is that the step relation is deliberately partial: it omits any step that would be illegal under the code consumer's safety policy. For example, suppose in some state (r, m) the program counter points at a instruction that would, if executed, load from an address outside the region permitted by the policy. Then, by the design of our step relation, there will not exist r', m' such that $(r, m) \mapsto (r', m')$. That is, the state (r, m) is *stuck* if it has no successor state in the $\mapsto$ relation.

In order to keep the model simple, we won't represent the notion of safely halting with a result—the only safe computations are those that continue forever. A safe state is one that cannot evaluate to a stuck state,

$$\text{safe}(r, m) \equiv \forall r', m'.\ (r, m) \mapsto^* (r', m') \Rightarrow \exists r'', m''.\ (r', m') \mapsto (r'', m'')$$

We say that a machine state (r, m) is *safe to execute for k steps* if it cannot get stuck within k instructions:

$$\text{safen}(k, r, m) \equiv \forall j < k\ \forall (r', m').\ (r, m) \mapsto^j (r', m') \Rightarrow \exists r'', m''.\ (r', m') \mapsto (r'', m'')$$

We write $m \sqsubseteq m'$ to mean that one partial memory approximates another,

$$m \sqsubseteq m' \equiv \forall x \in \text{dom}(m).\ x \in \text{dom}(m') \wedge m(x) = m'(x)$$

Sometimes we will want to talk about the safety of partial memories, that is, partial functions from addresses to integers. We can view a partial memory as an underspecified total memory, and it will be safe if every possible extension of it is safe.

$$\text{safen}_p(k, r, m) \equiv \forall m'.\ m \sqsubseteq m' \Rightarrow \text{safen}(k, r, m')$$

6. SETS OF INDEXED VALUES

Unlike values in lambda calculus, von Neumann values are not identified with terminated computations. A value is a data structure in memory, with a root pointer.

Just as in our λ-calculus model, a *type* is a set of indexed values $\{\langle k, m, x\rangle\}$ where k is an approximation index, m is a partial memory, and x is an integer (perhaps the root pointer of a data structure). Unlike the λ model, there are no expressions that are not values, since we are dealing with machine states. Therefore we have the correspondence,

$$(m, x) :_k \tau \equiv \langle k, m, x\rangle \in \tau$$

Intuitively, $(m, x) :_k \tau$ means that the data structure (m, x) approximately belongs to τ; if a continuation of type $\tau \to \bot$ is applied to (m, x), then the machine will not get stuck within k steps.

We say that a set of indexed values is a valid *type* if it is closed under extension of the memory and under decreasing index:

$$\text{type}(\tau) \equiv \forall m, m', x, j, k.m \sqsubseteq m' \wedge j \leq k \wedge \langle k, m, x\rangle \in \tau \Rightarrow \langle j, m', x\rangle \in \tau$$

As explained by Appel and Felty [2000], closure under extension of the memory is necessary so that the program can allocate and initialize a new value while preserving existing typing judgements about old values.

As in our λ-calculus model, we define an approx operator on types,

$$\text{approx}(k, \tau) = \{\langle j, m, x\rangle \mid j < k \wedge \langle j, m, x\rangle \in \tau\}$$

and we say that a type constructor F is well founded if

$$\forall \tau.\ \text{type}(\tau) \Rightarrow (\text{type}(F\tau) \\ \wedge \forall k.\text{approx}(k+1, F\tau) = \text{approx}(k+1, F(\text{approx}(k, \tau))))$$

Similarly, F is nonexpansive if

$$\forall \tau.\ \text{type}(\tau) \Rightarrow (\text{type}(F\tau) \wedge \forall k.\text{approx}(k, F\tau) = \text{approx}(k, F(\text{approx}(k, \tau))))$$

A type environment Φ or Γ is a finite map from integers to types. We will use Φ to specify *local invariants* that give the types of (some subset of) the registers at a certain program point, and Γ to specify the *global invariant* that gives the types of various program-counter locations in the program code.

We define $(m, f) :_k \Phi$ ("a function f satisfies Φ to approximation k") to mean,

$$(m, f) :_k \Phi \equiv \forall x \in \text{dom}(\Phi).\ (m, f(x)) :_k \Phi(x)$$

Type environments are used for two purposes: to summarize the types of the contents of machine registers (in which case f will be a register bank r), and to summarize the types of all entry points (machine-code addresses) of the program (in which case f will be the identity function, and we will typically write $(m, \text{id}) :_k \Gamma$).

A valid type environment is composed of valid types:

$$\text{typenv}(\Phi) \equiv \forall x \in \text{dom}(\Phi).\ \text{type}(\Phi(x))$$

$$\begin{aligned}
\textbf{int} &= \{\langle k, m, x\rangle \mid \text{true}\} \\
\textbf{const}(n) &= \{\langle k, m, x\rangle \mid x = n\} \\
\textbf{ref}(\tau) &= \{\langle k, m, x\rangle \mid x \in \text{dom}(m) \wedge \forall j < k.\ \langle j, m, m(x)\rangle \in \tau\} \\
\sigma \cap \tau &= \sigma \cap \tau \\
\sigma \cup \tau &= \sigma \cup \tau \\
\exists F &= \{\langle k, m, x\rangle \mid \exists \alpha.\ \text{type}(\alpha) \wedge \langle k, m, x\rangle \in F(\alpha)\} \\
\textbf{codeptr}(\Phi) &= \{\langle k, m, x\rangle \mid \forall j, r', m' \\
&\qquad m \sqsubseteq m' \wedge \text{dom}(m') = \text{alloc}(r', m') \wedge j < k \wedge r'(\text{pc}) = x \wedge (m', r') :_j \Phi \\
&\qquad \Rightarrow\ \text{safen}_p(j, r', m')\} \\
\mu F &= \{\langle k, m, x\rangle \mid \langle k, m, x\rangle \in F^{k+1}\bot\}
\end{aligned}$$

Any value x can be seen as a machine integer (regardless of the memory m that accompanies it). Intersection (respectively, union) types are defined via intersection (resp., union) of sets.

THEOREM 38. *Each of our types is a valid type*:

a. $\text{type}(\textbf{int})$.
b. $\text{type}(\textbf{const}(n))$.
c. $\text{type}(\tau) \Rightarrow \text{type}(\textbf{ref}(\tau))$.
d. $\text{type}(\sigma) \wedge \text{type}(\tau) \Rightarrow \text{type}(\sigma \cap \tau))$.
e. $\text{type}(\sigma) \wedge \text{type}(\tau) \Rightarrow \text{type}(\sigma \cup \tau))$.
f. $\text{nonexpansive}(F) \Rightarrow \text{type}(\exists F)$.
g. $\text{type}(\tau) \Rightarrow \text{type}(\textbf{codeptr}(\tau))$.
h. $\text{wellfounded}(F) \Rightarrow \text{type}(\mu F)$.

THEOREM 39. *The following typing lemmas hold*:

$$\frac{}{(m, x) :_k \textbf{int}} \qquad \frac{}{(m, x) :_k \textbf{const}(x)}$$

$$\frac{x \in \text{dom}(m) \quad (m, m(x)) :_{k-1} \tau}{(m, x) :_k \textbf{ref}(\tau)} \qquad \frac{(m, x) :_k \textbf{ref}(\tau)}{x \in \text{dom}(m) \quad (m, m(x)) :_{k-1} \tau}$$

$$\frac{\text{wellfounded}(F) \quad (m, x) :_k F(\mu F)}{(m, x) :_k \mu F} \qquad \frac{\text{wellfounded}(F) \quad (m, x) :_k \mu F}{(m, x) :_k F(\mu F)}$$

A program p is a sequence of machine instructions at a specific place in memory, that is, it is a finite function from address to integer, where the integer codes for an instruction. Thus p is just a partial memory, and we can say that p is embedded in a memory m by writing $p \sqsubseteq m$.

At each point in the program there is a precondition, or invariant, such that if the registers and memory satisfy the precondition it is safe to execute the program. Necula [1997] would express these preconditions using types, for example, $r(1) : \tau_1 \wedge r(2) : \tau_2 \wedge r(5) : \tau_5$.

But this is like saying that r satisfies a type environment $r:\Phi$, where $\Phi=\{1\mapsto\tau_1, 2\mapsto\tau_2, 5\mapsto\tau_5\}$. And the statement that this is the precondition of location l is the same as l : **codeptr**(Φ), that is, it is safe to execute from location l as long as the registers satisfy Φ.

For the remainder of this section we make the simplifying assumption that the program p contains only instructions, not data structures. Thus, for all l in the domain of p, $\Gamma(l)=$ **codeptr**(Φ_l) for some Φ_l. The statement that all the locations in the program have their respective codeptr types,

$$\forall l \in \mathrm{dom}(p).\ (p,l) : \mathbf{codeptr}(\Phi_l)$$

is the same as the statement that $(p,\mathrm{id}):\Gamma$, and id is the identity function; the identity function because here we are not reasoning about the *contents* of the ith register, but the *address* of the ith program location.

Scenario. A host computer wishes to run untrusted programs p. It will do so by (1) loading p into memory at address l_0, (2) setting register 1 to contain an integer argument, (3) setting register 6 to point to the beginning of the program's heap space, and (4) jumping to location l_0. The host's safety policy is that under these conditions, the initial state (r,m) must be provably safe; it will check the proof before doing steps 1–4.

The scenario assures of the initial state (r,m) that (1) $p\sqsubseteq m$, (2) r_1 is an integer, (3) alloc(r,m) properly describes the allocated locations, and (4) $r(\mathrm{pc})=l_0$.

THEOREM 40. *Let program p have entry point l_0 with formal parameters Φ_0. If p is loaded in memory m; the program counter is set to location l_0; the program satisfies invariants Γ, where $\Gamma(l_0)=$* **codeptr**(Φ_0)*; the register bank r satisfies Φ_0; then the program is safe*:

$$\frac{p\sqsubseteq m \quad r(\mathrm{pc})=l_0 \quad \forall k.(p,\mathrm{id}):_k \Gamma \quad \Gamma(l_0)=\mathbf{codeptr}(\Phi_0) \quad \forall k.(m,r):_k \Phi_0}{\mathrm{safe}(r,m)}$$

PROOF. For any k, the premise $(p,\mathrm{id}):_k \Gamma$ gives us $l_0 :_k$ **codeptr**(Φ_0). Then safen$_p(k,r,m)$ follows from the definition of **codeptr**(Φ_0). □

In our scenario, $\Phi_0(1)=\mathrm{int}$ and for $i\neq 1$, $\Phi_0(i)=\top$. By our definitions of int and $\top=\mathrm{int}$, $(m,r):_k \Phi_0$ is trivially satisfied. The only nontrivial premise of Theorem 40 is $\forall k.(p,\mathrm{id}):_k\Gamma$; to prove this we define some properties of program points.

Definition 41. A machine state (r,m) satisifies the precondition at l of program p to approximation k, if the program counter is at l, the local invariant Φ is satisfied, and the *alloc* function properly describes the set of allocated locations:

$$\begin{aligned}&\mathrm{sat\_precond}(p,\Gamma,l,k,r,m)\equiv\\ &\quad p\sqsubseteq m \ \wedge\ r(\mathrm{pc})=l \ \wedge\ \Gamma(l)=\mathbf{codeptr}(\Phi)\wedge(m,r):_k\Phi\wedge\mathrm{dom}(m)=\mathrm{alloc}(r,m)\end{aligned}$$

LEMMA 42. *If a program satisfies* Γ *to degree* k, *and some state satisfies a precondition in* Γ *to degree* $k-1$, *then it is safe for* $k-1$ *steps*:

$$\frac{(p, \mathrm{id}) :_k \Gamma \quad \mathrm{sat\_precond}(p, \Gamma, l, k-1, r, m)}{\mathrm{safen}_p(k-1, r, m)}$$

PROOF. From the definition of **codeptr**, taking $r' = r$, $m' = m$. □

Definition 43. A program p is *safe at* l with respect to global invariant Γ if, whenever the program satisifies Γ, and the state satisfies precondition at l to approximation k, it can take one step to a state that satisfies the precondition at some l' to approximation $k-1$:

$$\begin{array}{l}\mathrm{safe\_at}(p, \Gamma, l) \equiv \\ \quad \forall r, m, k.\ (p, \mathrm{id}) :_k \Gamma \Rightarrow \\ \qquad \mathrm{sat\_precond}(p, \Gamma, l, k, r, m) \Rightarrow \\ \qquad \exists! r', m', l'.\ (r, m) \mapsto (r', m') \wedge \mathrm{sat\_precond}(p, \Gamma, l', k-1, r', m')\end{array}$$

THEOREM 44. *If a program is safe_at every point*, *then it is safe*:

$$\frac{\forall l \in \mathrm{dom}(\Gamma).\ \mathrm{safe\_at}(p, \Gamma, l)}{\forall k.(p, \mathrm{id}) :_k \Gamma}$$

PROOF. By induction over k. Each $\Gamma(l)$ is a codeptr type, and these have the property that they accept any value to approximation zero; this proves the base case.

To prove the inductive case, assume $(p, \mathrm{id}) :_k \Gamma$. We will show $(p, \mathrm{id}) :_{k+1} \Gamma$ by proving for each l that $l :_{k+1} \Gamma(l)$, that is, $l :_{k+1} \mathbf{codeptr}(\Phi_l)$.

By the definition of **codeptr** this is,

$$\begin{array}{ll}\forall j, r, m. & p \sqsubseteq m \wedge \mathrm{dom}(m) = \mathrm{alloc}(r, m) \wedge j < k+1 \wedge r(\mathrm{pc}) = l \wedge (m, r) :_j \Phi_l \\ & \Rightarrow \mathrm{safen}_p(j, r, m)\end{array}$$

Pick arbitrary j, r, m and assume the premises $p \sqsubseteq m$, $\mathrm{dom}(m) = \mathrm{alloc}(r, m)$, $j < k+1$, $r(\mathrm{pc}) = l$, and $(m, r) :_j \Phi_l$. From this we have $\mathrm{sat\_precond}(p, \Gamma, l, j, r, m)$. Using the premise $\mathrm{safe\_at}(p, \Gamma, l)$ we know that $(r, m) \mapsto (r', m')$ such that $\mathrm{sat\_precond}(p, \Gamma, l', j-1, r', m')$. By Lemma 42, (r', m') is safe for $j-1$ steps, so (r, m) is safe for j steps. □

Thus, it suffices to prove safe_at l for each location $l \in \Gamma$. Since $\mathrm{dom}(\Gamma) = \mathrm{dom}(p)$, we know what instruction i is located at $p(l)$. We will need a proof tactic for each kind of instruction i.

LEMMA 45. *Suppose* $p(l)$ *is an integer that codes for the instruction* $r_3 \leftarrow m(r_4)$. *Suppose* $\Gamma(l) = \mathbf{codeptr}(\Phi_l)$ *and* $\Gamma(l+1) = \mathbf{codeptr}(\Phi_{l+1})$, *where*

$$\begin{aligned}\Phi_l &= \{1 : \mathbf{int}, 3 : \mathbf{int}, 4 : \tau_1 \times \tau_2\} \\ \Phi_{l+1} &= \{1 : \mathbf{int}, 3 : \tau_1, 4 : \tau_1 \times \tau_2\}\end{aligned}$$

Then $(\mathrm{safe\_at}(p, \Gamma, l))$

Informal argument. The precondition Φ_l of the instruction says, in effect, $r(1) : \mathbf{int}$, $r(3) : \mathbf{int}$, $r(4) : \tau_1 \times \tau_2$. The postcondition is $r(1) : \mathbf{int}$, $r(3) : \tau_1$, $r(4) : \tau_1 \times \tau_2$. The instruction fetches the first field of the pair. Since the type of the first field is τ_1, the destination register r_3 ends up with type τ_1.

PROOF. Assume $(p, \mathrm{id}) :_k \Gamma$, and sat_precond$(p, \Gamma, l, k, r, m)$.

Since a valid instruction p (and therefore in m) at location l, and (by sat$_{\mathrm{p}}$recond) we know $r(\mathrm{pc}) = l$, therefore the machine can execute a step, leading to a state (r', m').

By the definition of sat_precond, $p \sqsubseteq m$, and by the semantics of the instruction, $m = m'$. Thus, $p \sqsubseteq m'$, that is, executing this instruction does not overwrite the program.

Our instruction has modified neither $\mathrm{dom}(m)$ nor the registers within r that determine the alloc function; that is, $m = m'$, so $\mathrm{dom}(m) = \mathrm{dom}(m')$ and $\mathrm{alloc}(r, m) = \mathrm{alloc}(r', m')$. Therefore $\mathrm{dom}(m') = \mathrm{alloc}(r', m')$. But if we had an instruction that increased the allocated set (as described by Appel and Felty [2000]), this is where we would need to account for it.

Our example instruction is not a jump, so in the state r' we will have incremented the program counter by 1; that is, $r'(\mathrm{pc}) = 1 + r(\mathrm{pc}) = l'$. If it were a jump, then we would need to account for l' in a more sophisticated way than just $l' = l + 1$.

Finally, we must prove $(m', r') :_{k-1} \Phi_{l+1}$. That is, for all n in the domain of Φ_{l+1}, $\langle k - 1, m', r'(n)\rangle \in \Phi_{l+1}(n)$. The domain is just $\{1, 3, 4\}$; for $n = 1$ or 4 the proposition is trivial, since $\langle k, m, r(n)\rangle \in \Phi_l(n)$, $\Phi_l(n) = \Phi_{l+1}(n)$, $m = m'$, $r'(n) = r(n)$, and types are closed under decreasing index.

To prove $\langle k - 1, m', r'(3)\rangle \in \tau_1$, we work as follows. The premise $(m, r) :_k \Phi_l$ implies $\langle k, m, r(4)\rangle \in \tau_1 \times \tau_2$. By the definition of $\times$, $\langle k, m, r(4)\rangle \in \mathbf{ref}(\tau_1)$. By the definition of **ref**, $\langle k-1, m, m(r(4))\rangle \in \tau_1$. By the semantics of the fetch instruction, $r'(3) = m(r(4))$, so $\langle k - 1, m, r'(3)\rangle \in \tau_1$. Since $m = m'$, $\langle k - 1, m', r'(3)\rangle \in \tau_1$.

Therefore sat_precond$(p, \Gamma, l', k - 1, r', m')$. □

To prove a program p safe, we must have a battery of lemmas such as Lemma 45, and we must find the invariant Γ suitable for p. Neither of these is easy. We believe that the best way to succeed is to make a semantic model of Typed Assembly Language [Morrisett et al. 1998b], and then the compiler that produces p can also produce Γ. Such a semantic model would rely on the indexed model of recursive types that we have demonstrated here, but is beyond the scope of the current paper.

The reasoning in the proof of Lemma 45 is similar to what proof-carrying code systems do already: a combination of types (in the local invariants) and dataflow (to model instruction semantics) leads to a proof that the local invariant at location l naturally leads to the invariant at $l + 1$. The main difference is that we don't assume the typing rules as axioms of our system, but model the types within a more primitive logic and prove the rules as derived lemmas.

A natural generalization of our technique is to let $\mathrm{dom}(\Gamma)$ be only a subset of program locations in p, for example, one Φ at the entrance of each basic block. Then we need to show that if Φ_l holds, there is some sequence of n instructions (the entire basic block) that can be executed, leading to Φ_{l+n} (or to some other

location, if there has been a jump) whose invariant is then satisfied to at least degree $k - n$.

7. FIRST-CLASS FUNCTIONS

In a source language with first-class functions, the result of an expression can be a function value, which can be bound to a variable, stored into a data structure, and eventually applied to an argument. In a conventional translation to machine language, we will see the address of a segment of machine code being bound to a variable, stored into a data structure, and eventually jumped to (with arguments in the appropriate registers). In languages with higher-order functions implemented as closures, the machine-code pointers are still there, hidden inside the closures.

A type system for proof-carrying code must account for function values. Appel and Felty [2000] give a type system which includes function values (through a codeptr type similar in spirit to the one we have presented here) and covariant recursive types (not the general recursive types we have presented here). They also sketch a proof method for using these types to prove safety of programs.

The problem is that their proof method is too weak to accommodate first-class function values: it can handle application of first-class functions but not creation of them. No formal result in their paper is (known to be) wrong, but where they appear to imply that their method can accommodate function-pointers, they are mistaken. The example that follows cannot be typed in their system—in particular, at the instruction $r_2 \leftarrow 102$ it cannot be proved that $102 : \mathbf{int} \rightarrow \mathbf{int}$.

The problem is that their induction is forward, over execution steps since the beginning of the program. In contrast, the proof method using indexed types, as presented in the previous section, is by induction over future execution steps. Intuitively, codeptr values are (first-order) continuations, so it is natural that reasoning about future execution is the right way to proceed. And indeed, our indexed-type method is strong enough to handle programs with function pointers.

We will show an example, using a short machine-language program that puts a function-pointer into a register, then calls the function. In this example we use a very simple-minded notion of continuation type—$\mathbf{cont}(\tau)$, which is a continuation accepting a return-value of type τ in register 1,

$$\mathbf{cont}(\tau) = \mathbf{codeptr}\{r_1 : \tau\}$$

and an equally simple notion of function type, that is,

$$\tau_1 \rightarrow \tau_2 = \mathbf{codeptr}\{r_1 : \tau_1, r_7 : \mathbf{cont}(\tau_2)\}$$

This means that the formal parameter (of type τ_1 arrives in register 1, and the return address (of type $\mathbf{cont}(\tau_2)$ arrives in register 7. Return values (of type τ_2 are passed back in register 1. We ignore here the problem of stacking return addresses for nested calls, which is treated in depth elsewhere [Morrisett et al. 1998a].

Our program (with local invariants Φ) is

l	$p(l)$	Φ_l
100 :		$\{\}$
	$r_2 \leftarrow 102$	
101 :		$\{r_2 : \mathbf{int} \rightarrow \mathbf{int}\}$
	jump 104	
102 :		$\{r_1 : \mathbf{int}, r_7 : \mathbf{cont}(\mathbf{int})\}$
	$r_1 \leftarrow r_1 + 1$	
103 :		$\{r_1 : \mathbf{int}, r_7 : \mathbf{cont}(\mathbf{int})\}$
	jump r_7	
104 :		$\{r_2 : \mathbf{int} \rightarrow \mathbf{int}\}$
	$r_1 \leftarrow 3$	
105 :		$\{r_1 : \mathbf{int}, r_2 : \mathbf{int} \rightarrow \mathbf{int}\}$
	$r_7 \leftarrow 107$	
106 :		$\{r_1 : \mathbf{int}, r_2 : \mathbf{int} \rightarrow \mathbf{int}, r_7 : \mathbf{cont}(\mathbf{int})\}$
	jump r_2	
107 :		$\{r_1 : \mathbf{int}\}$
	jump 107	

Instruction 100 moves the function-pointer 102 into r_2, then jumps to 104. Instruction 104 marshals the argument 3 and return address 107 into registers r_1 and r_7, then jumps to the function-pointer. Instruction 107 (safely) infinite-loops.

We construct the global invariant Γ from the Φ functions shown in the table.

LEMMA 46. *This program is safe_at* $l = 100$.

PROOF. Most of the necessary conclusions are trivial. Certainly if $p \sqsubseteq m$, then $m(100)$ contains the instruction $r_2 \leftarrow 102$, so $(r, m) \mapsto (r', m')$ with $r'(2) = 102$ and $r'(\mathrm{pc}) = 101$. Certainly $\Gamma(101) = \mathbf{codeptr}(\Phi_{101}) = \mathbf{codeptr}\{r_2 : \mathbf{int} \rightarrow \mathbf{int}\}$. Since $m = m'$ and the predicate $\mathrm{alloc}(r, m)$ is independent of $r(2)$ and $r(\mathrm{pc})$, we have $p \sqsubseteq m'$ and $\mathrm{dom}(m') = \mathrm{alloc}(r', m')$.

Finally, we must show $(m', r') :_{k-1} \Phi_{101}$. By hypothesis, $(p, \mathrm{id}) :_k \Gamma$, so $\forall x \in \mathrm{dom}(p).\, [0](p, x) :_k \Gamma(x)$. Thus, $(p, 102) :_k \mathbf{codeptr}(\Phi_{102})$. But $\mathbf{codeptr}(\Phi_{102}) = \mathbf{codeptr}(\{r_1 : \mathbf{int}, r_7 : \mathbf{cont}(\mathbf{int})\}) = \mathbf{int} \rightarrow \mathbf{int}$. Thus,

$$(p, 102) :_k \mathbf{int} \rightarrow \mathbf{int}$$

Since $r'(2) = 102$ and types are closed under $\sqsubseteq$ and under decreasing k, we have

$$(m', r'(2)) :_{k-1} \mathbf{int} \rightarrow \mathbf{int}$$

Since $\Phi_{101} = \{r_2 : \mathbf{int} \rightarrow \mathbf{int}\}$ we have

$$(m', r') :_{j-1} \Phi_{101}$$

□

8. RELATED WORK

Appel and Felty [2000] present a type system that defines a semantics for types on von Neumann machines with higher order types and monotone recursive

types. However, their proof method involves establishing program invariants by induction over steps of computation. The classical program-invariant method can not handle assignments of the form $x = f$ where x is a program variable and f is a (higher order) procedure constant (or instruction pointer). Here we give a type semantics that includes general recursive types and give a proof method approriate for higher order program invariants. The proof method is analogous to a mutual recursion fixed point rule similar to the λ-calculus fixed point rule mentioned in the introduction.

Assignments of the form $x = f$ where f is an object (as opposed to a procedure) are handled in a classical program-invariant style in a (type-specialized) PCC system for Java developed by Colby et al. [2000]. Program-invariant safety proofs for object-oriented programs can be interpreted as control-flow analyses—each method invocation transfers control to a known set of possible instruction locations. Higher order type-theoretic methods, such as the one we present in this paper, seem more general than first order control-flow methods, for example, type theoretic methods easily handle the polymorphic case.

Each of our types contains more (operational) information than types used in other semantics such as D_∞ models and the ideal model of MacQueen et al. [1986]. To strip away the extra information from an indexed type, one can take the limit (or infinite intersection) over k:

$$\text{strip}(\tau) = \{v \mid \forall k.\, \langle k, v\rangle \in \tau\}$$

An analogous "strip" operator can be defined for indexed PER types.

One implication of this extra internal structure is that an indexed type can distinguish (just a little bit) between equivalent expressions, depending on the efficiency (in execution steps) of the computations. For example, take the expressions $e_1 = \mathbf{0}$ and $e_2 = (\lambda x.x)\mathbf{0}$. We have $e_2 :_1 \mathbf{int} \to \mathbf{int}$, but not $e_1 :_1 \mathbf{int} \to \mathbf{int}$. However, neither $e_2 : \mathbf{int} \to \mathbf{int}$ nor $e_1 : \mathbf{int} \to \mathbf{int}$, since as we refine the approximation we can detect that the expressions are not functions.

THEOREM 47 (METRIC SPACES). *Well founded type constructors are* contractive *in the metric-space sense of MacQueen et al.* [1986]. *Therefore our μ operator is a construction of the fixed point that they prove must exist.*

PROOF. Use the metric $|\tau_1 - \tau_2| = 2^{-\text{nearness}(\tau_1,\tau_2)}$, where the nearness$(\tau_1, \tau_2)$ is the least k such that approx$(k, \tau_1) \neq$ approx(k, τ_2). □

Still, they are proving the existence of fixed points directly on the "stripped" types, which we do not do. On the other hand, they model only membership, whereas our approach easily generalizes to model equivalence.

Information systems. Winskel [1993] builds a model of recursive types based on *information systems*, which account for the size of proofs. He constructs well-founded recursions based on this size measure. This is a reasonable way to proceed, at least for lambda-calculus, but constructing the mathematical infrastructure of the theory of information systems leads to a proof that is significantly larger overall than ours.

Recursion-theoretic semantics. Mitchell and Viswanathan's [1996] PER semantics is powerful and expressive, but it relies on many "elementary" results of recursion theory. Building a machine-checked proof of these results for a real machine architecture would require a very large implementation effort, and for this reason the recursion-theoretic approach is not attractive.

Compactness of evaluation. The notion of *minimal invariance*—as defined by Pitts [1996] and adapted by Birkedal and Harper [1997] to an operational setting—provides a relational interpretation of general recursive types. Like other earlier approaches, these approaches treat terms extensionally and hence appear to be fundamentally different from our approach. We have not investigated generalizing our approach to arbitrary logical relations, but the ease with which our indexed-sets proof generalized to indexed-PERs is a hint that such generalizations should be possible.

9. CONCLUSION

We have presented a direct construction of general recursive types that is well suited for "implementation" as a machine-checked proof in a von Neumann setting. No significant libraries of mathematics are required as support. In contrast, previous PER models of computable functions use large bodies of computability theory, such as simulation theorems. Metric-space models use the theory of complete metric spaces (Cauchy sequences) and the Banach fixed point theorem. We have "implemented" a machine-checked proof of Theorems 38 and 39 in about 2000 lines of Twelf [Pfenning and Schürmann 1999] code, using the logic described by Appel and Felty [2000].

Actually, the theory of complete metric spaces is not so hard to implement in higher-order logic; the first author (working with Amy Felty) built most of an implementation, in preparation for a machine-checked model of types based on MacQueen et al. [1986]. The problem, however, was in finding an appropriate metric for computation on a von Neumann machine. This paper demonstrates the metric, but in doing so, it avoids the need for metric spaces at all.

Our model has a unary (type membership only) variant and a PER (partial equivalence relation) variant, so it is expressive enough for a wide variety of applications.

The key feature of our model is that it reasons inductively about the number of future computation steps. Thus it is well suited to modelling type systems that use continuations, which are abstractions of future computations.

REFERENCES

APPEL, A. W. AND FELTEN, E. W. 2001. Models for security policies in proof-carrying code. Tech. Rep. TR-636-01, Computer Science, Princeton University.

APPEL, A. W. AND FELTY, A. P. 2000. 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*. ACM Press, New York, NY, 243–253.

BIRKEDAL, L. AND HARPER, R. 1997. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*. Springer-Verlag, Berlin.

COLBY, C., LEE, P., NECULA, G. C., BLAU, F., CLINE, K., AND PLESKO, M. 2000. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '00*). ACM Press, New York, NY.

CONSTABLE, R. L., ALLEN, S. F., BROMLEY, H. M., CLEAVELAND, W. R., CREMER, J. F., HARPER, R. W., HOWE, D. J., KNOBLOCK, T. B., MENDLER, N. P., PANANGADEN, P., SASAKI, J. T., AND SMITH, S. F. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ.

HARPER, R., HONSELL, F., AND PLOTKIN, G. 1993. A framework for defining logics. *J. ACM 40*, 1 (Jan.), 143–184.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. 1986. An ideal model for recursive polymophic types. *Information and Computation 71*, 1/2, 95–130.

MICHAEL, N. G. AND APPEL, A. W. 2000. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*. Springer-Verlag, Berlin. LNAI 1831.

MILNER, R. 1977. Fully abstract models of typed λ-calculi. *Theoretical Computer Science 4*, 1–22.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *POPL '96*: *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 271–283.

MITCHELL, J. C. AND VISWANATHAN, R. 1996. Effective models of polymorphism, subtyping and recursion. In *23rd International Colloquium on Automata*, *Languages*, *and Programming*. Springer-Verlag, Berlin.

MORRISETT, G., CRARY, K., GLEW, N., AND WALKER, D. 1998a. Stack-based typed assembly language. In *ACM Workshop on Types in Compilation*. Springer-Verlag, Berlin. LNCS 1473.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998b. From System F to typed assembly language. In *POPL '98*: *25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York NY, 85–97.

NECULA, G. 1997. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 106–119.

NECULA, G. C. 1998. Compiling with proofs. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

PFENNING, F. 1994. Elf: A meta-language for deductive systems. In *Proceedings of the 12th International Conference on Automated Deduction*, A. Bundy, Ed. Springer-Verlag LNAI 814, Nancy, France, 811–815.

PFENNING, F. AND SCHÜRMANN, C. 1999. System description: Twelf—a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, Berlin.

PITTS, A. M. 1996. Relational properties of domains. *Information and Computation 127*, 2, 66–90.

SCHMIDT, D. A. 1986. *Denotational Semantics*: *A Methodology for Language Development*. Allyn and Bacon, Boston, MA.

SCOTT, D. S. 1976. Data types as lattices. *SIAM J. Comput. 5*, 3, 522–87.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages*: *An Introduction*. MIT Press, Cambridge, MA.

Received November 2000; revised July 2001; accepted September 2001