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

Andrew W. Appel
Princeton University

David McAllester
AT&T Research

November 21, 2000

Abstract

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.

1 Introduction

Proof-carrying code (PCC) [Nec97] 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.

In most current approaches to PCC [Nec97, MWCG98], 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 [AF00b] 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 [MA00] and safety policies [AF00a] are easily defined in higher-order logic. In foundational PCC the code provider must give both the executable code plus 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 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 the trusted base can be smaller: its trusted base consists 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 [HHP93, Pfe94].

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 have been given a semantics in terms of metric spaces [MPS86] and in terms of PER models of Turing machine computations [MV96]. The metric space approach is less powerful – 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. But a prelinary investigation by the first author found no obvious definition of an appropriate metric for types on von Neumann machines. On the other hand the Mitchell-Viswanathan model [MV96] is adaptable to von Neumann machines, but would require years of effort "implementating" machine-checked proofs of basic results in computability theory [AF01].

Our new semantics is a term model, with type judgements that are 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.

Our indexed type approach appears to be novel in that it is intensional. In an intensional system two denotationally equivalent functions can be treated differently. In particular, in our system equivalent functions with different running times will satisfy different indexed type judgements. The semantic treatments of recursive types mentioned above are all extensional — equivalent terms are treated equivalently.

Although we do not prove the result here, 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, here we take a different approach to proving that typability implies safety.

Following [AF00b] we take a semantic approach. 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, i.e., if the premises are true then the conclusion is true. This ensures that derivable type judgements are true and 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 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 $\to$ 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.

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

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

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

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{(\lambda x.e_1)\ e_2 \mapsto (\lambda x.e_1)\ e_2'}$$

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

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

Figure 1: Small step semantics

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 $e \mapsto e_1 \mapsto \ldots \mapsto e_j$ where e_j is e'. We write $e \mapsto^* e'$ if $e \mapsto^j e'$ for some $j \geq 0$. 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.

$$\Gamma \models x : \Gamma(x) \qquad \Gamma \models 0 : \mathrm{int}$$

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

$$\frac{\Gamma \models e_1 : \alpha \quad \Gamma \models e_2 : \beta}{\Gamma \models \langle e_1,\ e_2\rangle : \alpha \times \beta}$$

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

$$\frac{\Gamma \models e : \mu F}{\Gamma \models e : F(\mu F)} \qquad \frac{\Gamma \models e : F(\mu F)}{\Gamma \models e : \mu F}$$

Figure 2: Type Inference Lemmas

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 $\langle j, v\rangle \in \tau$. For any closed expression e and type τ we write $e :_k \tau$ if e is safe for k steps and if whenever $e \mapsto^j v$ for some value v with $j < k$ we have $\langle k - j, v\rangle \in \tau$; that is,*

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

where $\mathrm{nf}(e')$ *means that e' is a normal form — has no successor in the step relation.*

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.

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

Definition 2
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* $x \in \mathrm{dom}(\Gamma)$ *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*

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

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*.*

The remainder of this section is devoted to proving the type inference lemmas in figure 2. Each of these lemmas 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 and 2 immediately imply the following.

Lemma 3
If $\models e : \alpha$ *then* e *is safe.*

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 $\Gamma \models 0 : \mathrm{int}$, 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 4
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 5
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 definition 1 implies 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 6 (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 5. ■

Theorem 7 (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 6, 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 8
If α *and* β *are types then so is* $\alpha \times \beta$*.*

Lemma 9
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 5. 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 2 in the same manner that theorem 6 follows from lemma 5.

Lemma 10
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 2 in the same way that theorem 6 follows from lemma 5.

We have now proved all of the type inference lemmas except those for the type constructor μ. To understand some of the subtleties involved in the type constructor μ let Ω be the term $(\lambda x.xx)(\lambda x.xx)$. The derivation in figure 3 shows how to use the type lemmas in figure 2 to derive $\models\Omega : \perp$. By lemma 3 we then have that Ω is safe. In the derivation $\Lambda\alpha.\alpha\to\perp$ is the set functional mapping the set α to the set $\alpha\to\perp$. The proof that Ω is safe (shown in figure 3) relies on the type lemmas for μ in figure 2 which we now prove. We first observe the following lemma.

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, $\to$, and $\times$ are well founded.

Definition 11
The k-approximation *of a set is the subset of its elements whose index is less than k:*

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

We have that if α is a type then $\mathrm{approx}(k,\,\alpha)$ is a type. We now define a notion of 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 12
A well founded functional *is a function F from types to types such that for any type τ and $k \geq 0$ we have*

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

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 13
For F well founded and $j \leq k$,

$$(1)\qquad \mathrm{approx}(j,F^j(\tau_1)) = \mathrm{approx}(j,F^j(\tau_2))$$

$$(2)\qquad \mathrm{approx}(j,F^j(\tau)) = \mathrm{approx}(j,F^k(\tau))$$

Proof: (1) By induction.

$$\begin{array}{l}
\mathrm{approx}(0,F^j(\tau_1)) = \perp = \mathrm{approx}(0,F^j(\tau_2)).\\
\mathrm{approx}(j+1,F^{j+1}(\tau_1)) =\\
\mathrm{approx}(j+1,F(F^j(\tau_1))) =\\
\mathrm{approx}(j+1,F(\mathrm{approx}(j,F^j(\tau_1)))) =\\
\mathrm{approx}(j+1,F(\mathrm{approx}(j,F^j(\tau_2)))) =\\
\mathrm{approx}(j+1,F(F^j(\tau_2))) =\\
\mathrm{approx}(j+1,F^{j+1}(\tau_2)))
\end{array}$$

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

Theorem 14
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 $j \leq k$.

$$\begin{array}{ll}
\langle k,v\rangle \in \mu F & \\
\langle k,v\rangle \in F^{k+1}(\perp) & \text{by def'n of } \mu F\\
\langle j,v\rangle \in F^{k+1}(\perp) & \text{by def'n of type}\\
\langle j,v\rangle \in \mathrm{approx}(j+1,F^{k+1}(\perp)) & \text{by def'n of approx}\\
\langle j,v\rangle \in \mathrm{approx}(j+1,F^{j+1}(\perp)) & \text{by Lemma 13}\\
\langle j,v\rangle \in F^{j+1}(\perp) & \text{by def'n of approx}\\
\langle j,v\rangle \in \mu F & \text{by def'n of } \mu F
\end{array}$$

■

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

Figure 3: A derivation of $\models \Omega : \bot$

Lemma 15

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

Lemma 16
If F is well founded,

$$\begin{array}{ll} (a) & \text{approx}(k, \mu F) = \text{approx}(k, F^k \bot) \\ (b) & \text{approx}(k+1, F(\mu F)) = \text{approx}(k+1, F^{k+1} \bot) \end{array}$$

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

$$\begin{array}{ll} \langle j, v \rangle \in \text{approx}(k, \mu F) & \\ j < k \wedge \langle j, v \rangle \in \mu F & \text{by def'n of approx} \\ j < k \wedge \langle j, v \rangle \in F^{j+1} \bot & \text{by def'n of } \mu F \\ j < k \wedge \langle j, v \rangle \in \text{approx}(j+1, F^{j+1} \bot) & \text{by def'n of approx} \\ j < k \wedge \langle j, v \rangle \in \text{approx}(j+1, F^k \bot) & \text{by Lemma 13} \\ j < k \wedge \langle j, v \rangle \in F^k \bot & \text{by def'n of approx} \\ \langle j, v \rangle \in \text{approx}(k, F^k \bot) & \text{by def'n of approx} \end{array}$$

(b) Each of the following sets is equivalent.

$$\begin{array}{ll} \text{approx}(k+1, F^{k+1} \bot) & \\ \text{approx}(k+1, F(F^k \bot)) & \\ \text{approx}(k+1, F(\text{approx}(k, F^k \bot))) & \text{well-foundedness} \\ \text{approx}(k+1, F(\text{approx}(k, \mu F))) & \text{by } (a) \\ \text{approx}(k+1, F(\mu F)) & \text{well-foundedness} \end{array}$$

■

Lemma 17
If F is well founded,

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

Proof: Each of the following sets is equivalent

$$\begin{array}{ll} \text{approx}(k, \mu F) & \\ \text{approx}(k, F^k \bot) & \text{by Lemma 16a} \\ \text{approx}(k, F^{k+1} \bot) & \text{by Lemma 13} \\ \text{approx}(k, \text{approx}(k+1, F^{k+1} \bot)) & \text{by Lemma 15} \\ \text{approx}(k, \text{approx}(k+1, F(\mu F))) & \text{by Lemma 16b} \\ \text{approx}(k, F(\mu F)) & \text{by Lemma 15} \end{array}$$

■

Theorem 18
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 18 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, we should have $\alpha = \alpha \to \bot$. So 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 to determine if $\langle k, \lambda x.e \rangle \in \alpha$ it suffices to know whether $\langle j, v \rangle \in \alpha$ for $j < k$. More formally, functionals can be proved to be well founded using lemma 20 below.

Definition 19
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 [AF00b].

Lemma 20

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

$$\begin{aligned}&\text{approx}(k+1,\ H(\alpha)) =\\&\text{approx}(k+1,\ H(\text{approx}(k,\ \alpha))) =\\&\text{approx}(k+1,\ H(\text{approx}(k,\ \text{approx}(k+1,\ \alpha)))) =\\&\text{approx}(k+1,\ H(\text{approx}(k+1,\ \alpha)))\end{aligned}$$

b. Let I be $\Lambda\alpha.\alpha$.

$$\text{approx}(k,\ I(\alpha)) = \text{approx}(k,\ I(\text{approx}(k,\ \alpha)))$$

c. Let K be a constant function.

$$\text{approx}(k,\ K(\alpha)) = \text{approx}(k,\ K(\text{approx}(k,\ \alpha)))$$

d.

$$\begin{aligned}&\text{approx}(k,\ F(G(\alpha))) =\\&\text{approx}(k,\ F(\text{approx}(k,\ G(\text{approx}(k,\ \alpha))))) =\\&\text{approx}(k,\ F(G(\text{approx}(k,\ \alpha)))) =\end{aligned}$$

e. $\text{approx}(0, F(H(\alpha))) = \text{approx}(0, F(H(\text{approx}(0,\ \alpha)))$

$$\begin{aligned}&\text{approx}(k+1,\ F(H(\alpha))) =\\&\text{approx}(k+1,\ F(\text{approx}(k+1,\ H(\text{approx}(k,\ \alpha))))) =\\&\text{approx}(k+1,\ F(H(\text{approx}(k,\ \alpha)))) =\end{aligned}$$

$$\text{approx}(0,\ H(F(\alpha))) = \text{approx}(0,\ H(F(\text{approx}(0,\ \alpha)))$$

$$\begin{aligned}&\text{approx}(k+1,\ H(F(\alpha))) =\\&\text{approx}(k+1,\ H(\text{approx}(k,\ F(\text{approx}(k,\ \alpha))))) =\\&\text{approx}(k+1,\ H(F(\text{approx}(k,\ \alpha)))) =\end{aligned}$$

f. By the definition of $\to$ we have the following.

$$\text{approx}(k+1,\ \alpha \to \beta) = \text{approx}(k+1,\ \text{approx}(k,\ \alpha) \to \text{approx}(k,\ \beta))$$

This gives the following.

$$\begin{aligned}&\text{approx}(0,\ F(\alpha) \to G(\alpha)))) =\\&\text{approx}(0,\ F(\text{approx}(0,\ \alpha)) \to G(\text{approx}(0,\ \alpha))))\end{aligned}$$

$$\begin{aligned}&\text{approx}(k+1,\ F(\alpha) \to G(\alpha)) =\\&\text{approx}(k+1,\ \text{approx}(k,\ F(\alpha)) \to \text{approx}(k,\ G(\alpha))) =\\&\text{approx}(k+1,\ \text{approx}(k,\ F(\text{approx}(k,\ \alpha)))\\&\qquad\qquad \to \text{approx}(k,\ G(\text{approx}(k,\ \alpha)))) =\\&\text{approx}(k+1,\ F(\text{approx}(k,\ \alpha)) \to G(\text{approx}(k,\ \alpha)))\end{aligned}$$

g. By the definition of $\times$ we have the following.

$$\text{approx}(k+1,\ \alpha \times \beta) = \text{approx}(k+1,\ \text{approx}(k,\ \alpha) \times \text{approx}(k,\ \beta))$$

This gives the following.

$$\begin{aligned}&\text{approx}(0,\ F(\alpha) \times G(\alpha)))) =\\&\text{approx}(0,\ F(\text{approx}(0,\ \alpha)) \times G(\text{approx}(0,\ \alpha)))\end{aligned}$$

$$\begin{aligned}&\text{approx}(k+1,\ F(\alpha) \times G(\alpha)) =\\&\text{approx}(k+1,\ \text{approx}(k,\ F(\alpha)) \times \text{approx}(k,\ G(\alpha))) =\\&\text{approx}(k+1,\ \text{approx}(k,\ F(\text{approx}(k,\ \alpha)))\\&\qquad\qquad \times \text{approx}(k,\ G(\text{approx}(k,\ \alpha)))) =\\&\text{approx}(k+1,\ F(\text{approx}(k,\ \alpha)) \times G(\text{approx}(k,\ \alpha)))\end{aligned}$$

■

Lemma 21
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 $\tau \in$ type means, as usual, that τ is closed under decreasing index.

Theorem 22 (Typing rules for quantified types)

$$(a)\qquad \frac{\forall\tau\in\text{type}.\;\; F\,\tau\in\text{type}}{(\exists F)\in\text{type}\qquad (\forall F)\in\text{type}}$$

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

$$(c)\qquad \frac{\Gamma\models v:\exists F}{\exists\tau\in\text{type}.\,\Gamma\models v : F\tau}$$

$$(d)\qquad \frac{\forall\tau\in\text{type}.\,\Gamma\models v : F\tau}{\Gamma\models v:\forall F}$$

$$(e)\qquad \frac{\Gamma\models v:\forall F}{\forall\tau\in\text{type}.\,\Gamma\models v : F\tau}$$

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 usually involve the explicit management of a set of type variables. It should be possible to define an extended notion of semantic entailment $\Delta,\Gamma\models_k e:\tau$ to support this form of type checking.

But 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.

Conclusion. Any type constructor $\Lambda\alpha.\tau$, where τ is built from α and the operators $\mathbf{int},\top,\bot,\times,\to$ is either well founded or the identity. Thus all of the typing rules of Figure 2 are valid. By Lemma 3, 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 conventionally typed λ-expressions.

3 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 more perfect information hiding across interfaces. Readers not interested in per's can skip this section, as later sections do not depend on it.

Our indexed model extends easily to per (partial equivalence relation) 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

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

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

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

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

$$\begin{array}{l} \sigma_1\le\sigma_2 :_k \Gamma \;\equiv \\ \quad \mathrm{dom}\,\sigma_1 = \mathrm{dom}\,\sigma_2 = \mathrm{dom}\,\Gamma \wedge \forall x.\;\sigma_1(x)\le\sigma_2(x) :_k \Gamma(x) \end{array}$$

$$\begin{array}{l} \Gamma\models_k e\le f:\tau \;\equiv \\ \quad \forall\sigma_1,\sigma_2.\;\sigma_1\le\sigma_2 :_k \Gamma \Rightarrow \sigma_1(e)\le \sigma 2(f) :_k \tau \end{array}$$

$$\Gamma\models e\le f:\tau \;\;\equiv\;\; \forall k.\,\Gamma\models_k e\le f:\tau$$

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

Lemma 23
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 $(e_1f_1)\le(e_2f_2) :_k \beta$.

Proof: By analogy with the proof of Lemma 5. Both e_1 and e_2 are safe for k steps. If $e_1\mapsto^{j_1} v_1$ with $j<k$, then v_1 must be a lambda expression $\lambda x.e$ and $f_1\mapsto^* f_1'$ with $\langle k-j,\lambda x.e,f_1'\rangle\in\alpha\to\beta$. Hence, the application e_1e_2 either reduces for k steps without any top-level beta-reduction – in which case $e_1e_2\le f :_k \tau$ for any f and τ – or $(e_1e_2)\mapsto^{j_1} (\lambda x.e)e_2 \mapsto^{j_2} (\lambda x.e)v$ 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 $e[v/x] \le f[f_2/y] :_{k-j_1-j_2-1}$. Either $e[v/x]$ steps for another $k - j_1 - j_2 - 1$ – in which case $e_1 e_2$ has now stepped for k steps and $e_1 e_2 \le f :_k \tau$ 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, $e_1 e_2 \mapsto^{j_1+j_2+1+j_3} v_3$; but $f_1 f_2 \mapsto^* f_3$, with the required relation between v_3 and f_3. ■

Theorem 24 (Application)

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

Proof: By analogy with Theorem 6, but using Lemma 23. ■

Corollary 25 $\dfrac{\Gamma \models e_1 : \alpha \to \beta \quad \Gamma \models e_2 : \alpha}{\Gamma \models (e_1\ e_2) : \beta}$

Theorem 26 (Abstraction)

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

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

Corollary 27 $\dfrac{\Gamma[x := \alpha] \models e : \beta}{\Gamma \models \lambda x.e : \alpha \to \beta}$

Lemma 28
*If α and β are types (i.e., closed under decreasing index), then so are $\perp$, **int**, $\alpha \times \beta$, and $\alpha \to \beta$.*

Definitions, Lemmas, and Theorems 11–18 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 29
All the statements (a)–(g) of Lemma 20, and Lemma 21, hold for indexed-per type constructors.

Theorem 30
*Any type constructor F expressible in the "syntax" of constructors **int**, $\times, \to, \mu$ is well founded, so therefore $\mu F = F(\mu F)$.*

Lemma 31
If $\models e : \alpha$ then e is safe.

But in the per model, 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 $e \le f : \tau \wedge f \le e : \tau$.

Theorem 32 (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 24. ■

Theorem 33 (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 $\alpha \to \beta$. ■

Theorem 34 (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 24 (and a similar theorem for pairing) e is *applicatively equivalent* to f. Observational equivalence follows via a straightforward adaption (for this calculus) of Milner's Context Lemma [Mil77]. ■

4 Proof-carrying code

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) [AF00b, MA00].

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 $\mathbf{offset}(0, \mathbf{ref}(\sigma)) \cap \mathbf{offset}(1, \mathbf{ref}(\tau))$; this is 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 $(\mathbf{const}(0) \times \sigma) \cup (\mathbf{const}(1) \times \tau)$, 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.

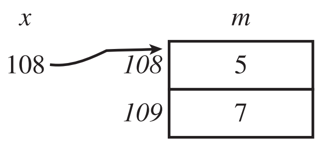
$\exists \alpha.\tau$ An existential type.

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 [MWCG98, AF00b].

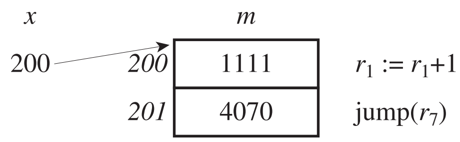
A *value* is a pair (m, x) where m is a finite partial function from integers to integers (a partial memory) and x is an integer (typically representing an address).[1]

[1] 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.

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 is 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, perhaps register $r(37)$ is the program counter, pc = 37. Then a machine state (r, m)in which we have just jumped to location 200 has the property $r(\mathrm{pc}) = 200$.

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 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 $\mathrm{alloc}(r, m)$ that takes a register bank and memory and returns a set of addresses (integers). An example of a simple alloc function is

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

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 $r(6)$.

We say that a machine state (r,m) is *stuck* if it has no successor state in the $\mapsto$ relation. A safe state is one that cannot evaluate to a stuck state,

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

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

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

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

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

Sometimes we will want to talk about the safety of partial memories, i.e. 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.

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

5 Sets of indexed values

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).[2] 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 $\tau \to$ Answer 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:

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

As explained by Appel and Felty [AF00b], 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

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

Similarly, F is nonexpansive if

$$\begin{aligned}&\forall \tau.\text{type}(\tau) \Rightarrow\\ &\quad (\text{type}(F\tau) \wedge\\ &\quad \forall k.\text{approx}(k,F\tau) = \text{approx}(k,F(\text{approx}(k,\tau))))\end{aligned}$$

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.

A mapping f satisfies Φ if

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

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, as we will explain).

A valid type environment is composed of valid types:

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

[2] The triples $\langle k,m,x\rangle$ do not correspond to the triples $\langle k,v,w\rangle$ used in Section 3. If we were designing the notation for more generality, we would write an indexed von Neumann per model with sets of $\langle k,\langle\langle m_1,x_1\rangle,\langle m_2,x_2\rangle\rangle\rangle$ where $v = \langle m_1,x_1\rangle$ and $w = \langle m_2,x_2\rangle$; then we could even generalize so that the single-argument "relation" $\langle m,x\rangle$ and the two-argument relation $\langle\langle m_1,x_1\rangle,\langle m_2,x_2\rangle\rangle$ could be uniformly indexed by k in all the definitions, lemmas, and proofs. We avoided this extravaganza of abstraction in the interests of readability.

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

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 35
Each of our types is a valid type:

a. type(**int**).

b. type(**const**(n)).

c. $\mathrm{type}(\tau) \Rightarrow \mathrm{type}(\textbf{ref}(\tau))$.

d. $\mathrm{type}(\sigma)\wedge\mathrm{type}(\tau) \Rightarrow \mathrm{type}(\sigma\cap\tau)$.

e. $\mathrm{type}(\sigma)\wedge\mathrm{type}(\tau) \Rightarrow \mathrm{type}(\sigma\cup\tau)$.

f. $\mathrm{nonexpansive}(F) \Rightarrow \mathrm{type}(\exists F)$.

g. $\mathrm{type}(\tau) \Rightarrow \mathrm{type}(\textbf{codeptr}(\tau))$.

h. $\mathrm{wellfounded}(F) \Rightarrow \mathrm{type}(\mu F)$.

Theorem 36
The following typing rules apply:

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

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

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

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

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

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. Following Necula [Nec97] we express these preconditions using types, e.g., $r(1) : \tau_1 \wedge r(2) : \tau_2 \wedge r(5) : \tau_5$. (We haven't formally defined unindexed typing judgements, so let's not assign too formal a meaning to this statement.)

But this is the same as 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 $l : \textbf{codeptr}(\Phi)$, that is, it is safe to execute from location l as long as the registers satisfy Φ.

The statement that all the locations in the program have their respective codeptr types,

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

is the same as the statement that $(p,\mathrm{id}) : \Gamma$, where $\Gamma(l) = \textbf{codeptr}(\Phi_l)$ for all l in the domain of p, 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.

To prove an initial machine state (r,m) safe, we will need premises of the form $p \sqsubseteq m$ (that is, a certain program is loaded in memory) and $r(\mathrm{pc}) = l_0$ (that is, the program counter is initially at a specified entry point). Then we will prove $\forall k.(p,\mathrm{id}) :_k \Gamma$ for some Γ such that $\Gamma(l_0) = \textbf{codeptr}(\Phi_{l_0})$; as we will explain, this will be by induction over k. Finally we will prove $\forall k.\,(p,r) :_k \Phi_{l_0}$, that is, the initial precondition of the program is met; this may be trivial if, for example, Φ_{l_0} is trivial. Then from the definition of codeptr, $\mathrm{safen}(j,r,m)$ for any k and any $j<k$, and thus $\mathrm{safe}(r,m)$.

Theorem 37 $\qquad \forall k.\ (p,\mathrm{id}) :_k \Gamma$

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 $(p,\mathrm{id}) :_k \Gamma \Rightarrow (p,\mathrm{id}) :_{k+1} \Gamma$ we work by cases; that is, we prove for each l in $\mathrm{dom}(p)$ that $l : \textbf{codeptr}(\Phi_l)$, or more precisely, $\langle k+1, p, l\rangle \in$

codeptr(Φ_l). By the definition of **codeptr** this is,

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

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 $(m', r') :_j \Phi_l$. What we must prove is that at location l in p there is some instruction i satisfying Lemma 38; an instance of this lemma must be proved for each l in the program. We discuss strategies for such proofs in the next section. Basically, the lemma says that starting from location l, the machine will execute at least one instruction and then satisfy Γ to approximation $k-1$; this is sufficient to prove that at l the machine satisfies Γ to approximation k.

Meanwhile, by the induction hypothesis, $\langle k, p, l'' \rangle \in$ **codeptr**$(\Phi_{l''})$. This, along with the conclusions of Lemma 38, proves $\mathrm{safen}_p(j-1, r'', m'')$, it is safe to execute $j-1$ instructions from (r'', m''). But the step $(r', m') \overset{i}{\mapsto} (r'', m'')$ means that from state (r', m') it must have been safe to execute at least j instructions; and this proves the desired conclusion, that $\langle k+1, p, l \rangle \in$ **codeptr**(Φ_l). ■

This proof has relied on details of p and Γ, through many instances (one for each l) of the following lemma:

Lemma 38

$$\frac{\begin{array}{cccc} p \sqsubseteq m' & \mathrm{dom}(m') = \mathrm{alloc}(r', m') & & \\ (p, \mathrm{id}) :_k \Gamma & j < k+1 & r'(\mathrm{pc}) = l & (m', r') :_j \Phi_l \end{array}}{\begin{array}{cc} \exists! \, r'', m'', l'', \Phi_{l''}. (r', m') \overset{i}{\mapsto} (r'', m'') & r''(\mathrm{pc}) = l'' \\ \Gamma(l'') = \mathbf{codeptr}(\Phi_{l''}) & p \sqsubseteq m'' \\ \mathrm{dom}(m'') = \mathrm{alloc}(r'', m'') & (m'', r'') :_{j-1} \Phi_{l''} \end{array}}$$

We will sketch instances of how the lemma can be proved, and then argue that these proofs are consistent with other formulations of proof-carrying code, so that we can expect this method to scale to real programs.

Example 39
*At location l there is an integer that codes for the instruction $r_3 \leftarrow m(r_4)$. $\Gamma(l) =$ **codeptr**(Φ_l) and $\Gamma(l+1) =$ **codeptr**(Φ_{l+1}), where*

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

The precondition Φ_l of the instructions says, in effect

$$r(1) : \mathbf{int} \qquad r(3) : \mathbf{int} \qquad r(4) : \tau_1 \times \tau_2$$

The postcondition is

$$r(1) : \mathbf{int} \qquad r(3) : \tau_1 \qquad r(4) : \tau_1 \times \tau_2$$

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.

We claim that Lemma 38 holds.

Proof: By the premises, $p \sqsubseteq m'$. Thus, the instruction at location l that was originally in the program p is still in memory (has not been overwritten) by the time we reach state (r', m').

Since a valid instruction i is in p' (and therefore in m') at location l, and (by a premise) $r'(pc) = l$, we know that the machine can execute a step, leading to a state (r'', m'').

Our example instruction i does not store into memory, so $m' = m''$; but if it were a store, premise $\mathrm{dom}(m') = \mathrm{alloc}(r', m')$. could help us prove that the address stored into is not within $\mathrm{dom}(m')$, and thus $m' \sqsubseteq m''$. Since $p \sqsubseteq m'$, we have $p \sqsubseteq m''$.

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 [AF00b]), 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 $(m'', r'') :_{j-1} \Phi_{l+1}$. That is, for all n in the domain of Φ_{l+1}, $\langle j-1, m'', r''(n) \rangle \in \Phi_{l+1}(n)$. The domain is just $\{1,3,4\}$; for $n = 1$ or 4 the proposition is trivial, since $\langle j, m', r'(n) \rangle \in \Phi_l(n)$, $\Phi_l(n) = \Phi_{l+1}(n)$, $m' = m''$, $r''(n) = r'(n)$, and types are closed under decreasing index.

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

Certain details that we omit in this paper, such as the axiomatization of the instructions, and the enforcement of memory safety such that only fetches from designated ranges of memory, are easily handled by the techniques shown in earlier papers [AF00b, MA00].

The reasoning in the proof of Example 39 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 $j-n$.

6 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 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. No formal result in their paper is (known to be) wrong, but they appear to imply that their method can accommodate function-pointers, they are mistaken. The problem is that their induction is forward, over execution steps since the beginning of the program. In contrast, the proof method presented 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 — $\mathbf{cont}(\tau)$ 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,

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

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 [MCGW98].

Our program (with local invariants Φ) is

l	$p(l)$	Φ_l
100 :		$\{\}$
	$r_2 \leftarrow 102$	
101 :		$\{r_2 : \mathbf{int} \to \mathbf{int}\}$
	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 :		$\{r_2 : \mathbf{int} \to \mathbf{int}\}$
	$r_1 \leftarrow 3$	
105 :		$\{r_1 : \mathbf{int},\, r_2 : \mathbf{int} \to \mathbf{int}\}$
	$r_7 \leftarrow 107$	
106 :		$\{r_1 : \mathbf{int},\, r_2 : \mathbf{int} \to \mathbf{int},\, r_7 : \mathbf{cont}(\mathbf{int})\}$
	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.

Example 40
Lemma 38 is provable for location $l = 100$.

Proof: Most of the necessary conclusions are trivial. Certainly if $p \sqsubseteq m'$, then $m'(100)$ still contains the instruction $r_2 \leftarrow 102$, so $(r', m') \mapsto (r'', m'')$ with $r''(2) = 102$ and $r''(\mathrm{pc}) = 101$. Certainly $\Gamma(101) = \Phi_{101} = \{r_2 : \mathbf{int} \rightarrow \mathbf{int}\}$. 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 $(m'', r'') :_{j-1} \Phi_{101}$. By a premise, $p :_k \Gamma$, so $\forall x \in \mathrm{dom}(p).\, (p, x) :_k \Gamma(x)$. 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

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

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

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

■

7 Related work

Appel and Felty's type system [AF00b] 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. [CLN$^+$00]. 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, e.g., 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, Plotkin, and Sethi [MPS86]. To strip away the extra information from an indexed type, one can take the limit (or infinite intersection) over k:

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

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 $e_2 :_1 \mathbf{int} \rightarrow \mathbf{int}$, but not $e_1 :_1 \mathbf{int} \rightarrow \mathbf{int}$. However, neither $e_2 : \mathbf{int} \rightarrow \mathbf{int}$ nor $e_1 : \mathbf{int} \rightarrow \mathbf{int}$, since as we refine the approximation we can detect that the expressions are not functions.

Theorem 41 (Metric spaces)
Well founded type constructors are contractive *in the metric-space sense of MacQueen, Plotkin, and Sethi. Therefore our μ operator is a construction of the fixed point that they prove must exist.*

Proof: Use the metric $|\tau_1 - \tau_2| = 2^{-\mathrm{nearness}(\tau_1, \tau_2)}$, where the $\mathrm{nearness}(\tau_1, \tau_2)$ is the least k such that $\mathrm{approx}(k, \tau_1) \neq \mathrm{approx}(k, \tau_2)$. ■

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

Recursion-theoretic semantics. Mitchell and Viswanathan's per semantics [MV96] is powerful and expressive, but it relies on many "elementary" results of recursion theory. It turns out in practice [AF01], that 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 [Pit96] and adapted by

Birkedal and Harper to an operational setting [BH97] — 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.

8 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 35 and 36 in about 2000 lines of Twelf [PS99] code, using the logic described by Appel and Felty [AF00b].

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, Plotkin, Sethi. But the problem 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

[AF00a] Andrew W. Appel and Edward W. Felten. Models for security policies in proof-carrying code. October 2000.

[AF00b] 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. ACM Press, January 2000.

[AF01] Andrew W. Appel and Amy P. Felty. Recursion-theoretic semantics for proof-carrying code. (in preparation), 2001.

[BH97] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*, 1997.

[CLN$^+$00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*. ACM Press, June 2000.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[MA00] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, June 2000.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *ACM Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[Mil77] Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymophic types. *Information and Computation*, 71(1/2):95–130, 1986.

[MV96] John C. Mitchell and Ramesh Viswanathan. Effective models of polymorphism, subtyping and recursion. In *23rd International Colloquium on Automata, Languages, and Programming*. Springer-Verlag, 1996.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, January 1998.

[Nec97] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[Pit96] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.