An Indexed Model of Impredicative Polymorphism and Mutable References

Amal Ahmed Andrew W. Appel Roberto Virga

Princeton University
{amal,appel,rvirga}@cs.princeton.edu

Abstract

We present a semantic model of the polymorphic lambda calculus augmented with a higher-order store, allowing the storage of values of any type, including impredicative quantified types, mutable references, recursive types, and functions. Our model provides the first denotational semantics for a type system with updatable references to values of impredicative quantified types. The central idea behind our semantics is that instead of tracking the exact type of a mutable reference in a possible world our model keeps track of the approximate type. While high-level languages like ML and Java do not themselves support storage of impredicative existential packages in mutable cells, this feature is essential when representing ML function closures, that is, in a target language for typed closure conversion of ML programs.

1 Introduction

Semantics of mutable references are hard [11, 17]. Recent possible-worlds models [1, 13] work well, but do not permit polymorphism. Semantics of impredicative polymorphism are also hard [9] and, as far as we know, there are no possible-worlds models of impredicative polymorphism. Combining mutable references with polymorphism can be extremely tricky. Our LICS'02 paper [2] showed how to model mutable references, and we claimed that we could handle impredicative polymorphism. We discovered, however, that we could mix them in all ways except for one: values of quantified types could not be stored in a mutable cell. But this mixture of features is necessary for typed closure conversion (type-preserving compilation of higher-order functional programming languages) [14], even without intensional type-passing [15]. There may, of course, be other, as yet undiscovered, ways of representing function closures, but for now it seems we need mutable references to impredicative quantified types.

For our Foundational Proof-Carrying Code project [3], we wish to construct type systems with soundness proofs that are machine-checkable in the simplest possible logic. Conventional syntactic type systems have proofs that are syntactic metatheorems by induction over proofs; to make such a proof machine-checkable requires a complicated and sophisticated checker such as the metatheory engine of the Twelf system [16]. Crary [10] has built a syntactic progress-and-preservation proof in the style of Harper [12] for a typed assembly language with mutable fields and impredicative polymorphism — the proof is checked by the Twelf metatheorem prover. In contrast, our semantic approach has been to define a type as a predicate on data structures, and a type constructor as a transformer of these predicates [4, 5, 2]. (In practice, a type can't be quite as simple as a predicate on terms; there must be auxiliary parameters that encode, in effect, a semantic domain construction.) Then each typing rule can be proved as a derived lemma in higher-order logic; these proofs are machine-checkable by a very simple and trustworthy program indeed [6]. The goal is to prove and use a type-soundness theorem: a given machine-language program type-checks; type-checking implies safety; so the program is safe.

Our prototype compiler [8] translates Core ML into Sparc machine language. The type-preserving translation of higher-order ML functions into function closures uses an existential quantifier to hide the type of the function environment [14]. This environment can contain other function-closures—that is, the instantiation of the existential can be by other existential types—so that the quantification must be impredicative. Of course, ML has mutable references, and a function-closure can be stored into a mutable reference, so references must be able to contain existentials and vice versa. Therefore we needed to construct the semantic model that we will describe in this paper, with all of these features as well as other features that we have previously described how to model: recursive types [5], heap allocation of data structures [4], address arithmetic, and so on.

In this paper, to keep the presentation simple, we will show only a model for λ-calculus. The machine-checked proof we are constructing (in higher-order logic represented in LF and type-checked by Twelf) is embedded in our pro-

totype PCC system for von Neumann machines (e.g., the Sparc); Appel and McAllester [5] show the similaries between models of types for λ-calculus and for von Neumann machines.

Our model, and proofs, follow as closely as possible the Appel and McAllester schema, so we will summarize it here. In that model, a type τ is a set of pairs $\langle k, v\rangle$, where k is a natural number and v is a (λ-calculus) value. The intuitive idea is 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. A typing judgment $e :_k \tau$ on a closed expression e means (by definition) that e can't get stuck within k steps, and furthermore if e reduces in $j < k$ steps to a value v then $\langle k-j, v\rangle \in \tau$. They call k the *approximation index* in such judgments.

Then they give a semantic definition of type-checking open expressions in a context, $\Gamma \vDash_k e : \tau$, and finally define $\Gamma \vDash e : \tau$ to mean $\forall k \geq 0.(\Gamma \vDash_k e : \tau)$. Using these definitions, they can prove conventional-looking rules such as

$$\frac{\Gamma \vDash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vDash e : \tau_1}{\Gamma \vDash (e_1\ e_2) : \tau_2}$$

directly from the definitions. Even the proofs regarding fold/unfold of contravariant equirecursive types are remarkably concise.

When we add mutable references, the small-step operational semantics is no longer of the form $e \mapsto e'$, but is $(M, e) \mapsto (M', e')$. In our new semantics, an approximate typing judgment is no longer just $v :_k \tau$ but $v :_{k,\Psi,M} \tau$. That is, to judge whether a value v is well typed, we must have a memory M in which to look up any location-labels that might appear within v, and we also need a *memory typing* Ψ that tells us about the types of those locations. In our new model, therefore, a type τ is a set of four-tuples $\langle k, \Psi, M, v\rangle$.

A most delicate part of the semantic construction is the fact that a type is a set of four-tuples, and the Ψ component of one of those four-tuples is itself a mapping from locations to types. If the construction were attempted naively, this would lead to a cardinality paradox. We avoid all paradoxes as follows. First, given $\langle k, \Psi, M, v\rangle$ and a location ℓ such that $\Psi(\ell) = \tau'$, we make sure that every element $\langle k', \Psi', M', v'\rangle \in \tau'$ has $k' < k$. Because of this inherent wellfoundedness, we can use a version of our previously described stratified Gödelization [2] to embed the entire model into ordinary higher-order logic.

Our previous stratified model of mutable references equipped each machine state (M, e) with a memory typing Ψ (called *allocset* a in that paper) that mapped each mutable location to its type. To avoid circularity, we had to ensure that all the Ψ's in the semantic definition of a type τ contained only strictly less complex types than τ. The third level in our hierarchy of types contained types such as $\mathbf{ref}\ (\mathbf{ref}\ (\mathbf{int}))$ but not $\mathbf{ref}\ (\mathbf{ref}\ (\mathbf{ref}\ (\mathbf{int})))$. For any (finite) type expression there was some level of the hierarchy powerful enough to contain it. Since all the type expressions in a well-typed monomorphic program are finite, we could find some level of the hierarchy strong enough to type each program.

However, in the presence of quantified types we encountered a problem. Consider, for example, the type $\exists\alpha.\mathbf{ref}\ (\alpha \times \alpha)$. We cannot predict how complex will be the type that instantiates α, so there is no finite level of the hierarchy that is guaranteed powerful enough. For example, if α is instantiated with $\tau = \mathbf{ref}^{50}(\mathbf{int})$, then the 43rd level of the hierarchy won't contain τ—only the 51st level (and above) will.

The solution, which we describe more rigorously in the next sections, is to use the approximation power of the indexed model. Suppose in some execution we are about to instantiate $\exists\alpha.\mathbf{ref}\ (\alpha \times \alpha)$ with $\mathbf{ref}^{50}(\mathbf{int})$, but we intend to run the program for only 30 more execution steps. Then it's all the same whether we instantiate with $\mathbf{ref}^{50}(\mathbf{int})$ or with $\mathbf{ref}^{30}(\bot)$, since in 30 execution steps the program cannot dereference more than 30 references. Therefore, if we intend to run the program for only k steps, it suffices to use the kth level of the hierarchy.

2 Indexed Types for the Lambda Calculus

Syntax. The language we shall consider is the polymorphic lambda calculus augmented with mutable references, existential and recursive types, and the constant $\mathbf{0}$. Cartesian products and other standard constructions are easy to add to the model we will present. The syntax of lambda terms is given by the following grammar.

Expressions	e	$::=$	$x \mid \ell \mid \mathbf{0} \mid \lambda x.e \mid (e_1\ e_2) \mid$
			$\mathtt{new}(e) \mid !\,e \mid e_1 := e_2 \mid \Lambda.e \mid e[] \mid$
			$\mathtt{pack}\,e \mid \mathtt{open}\,e_1\,\mathtt{as}\,x\,\mathtt{in}\,e_2$
Values	v	$::=$	$\ell \mid \mathbf{0} \mid \lambda x.e \mid \Lambda.e \mid \mathtt{pack}\,v$

We use the meta-variable x to range over a countably infinite set of *variables* and the meta-variable ℓ to range over a countably infinite set of *locations*. A term v is a *value* if it is a location ℓ, the constant $\mathbf{0}$, a term or type abstraction, or an existential package, and if it contains no free term variables x.

This syntax is slightly unconventional: in most presentations the polymorphic operators $\Lambda\alpha.e$ and $e[\tau]$ and the existential operators `pack` and `open` mention types syntactically. We wish to give purely semantic (not syntactic) typings to untyped lambda calculus, so we omit all types from our syntax. We let the vestigial operators remain in the untyped syntax to simplify the presentation.

Operational Semantics and Safety. A *memory* M is a mapping from locations to closed values. The small-step semantics (see Figure 1) is given by an abstract machine. The

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

$$\frac{(M,\, e) \mapsto (M',\, e')}{(M,\, \mathtt{new}(e)) \mapsto (M',\, \mathtt{new}(e'))} \qquad \frac{\ell \notin dom(M)}{(M,\, \mathtt{new}(v)) \mapsto (M\,[\ell := v],\, \ell)} \qquad \frac{(M,\, e) \mapsto (M',\, e')}{(M,\, !\, e) \mapsto (M',\, !\, e')} \qquad \frac{\ell \in dom(M)}{(M,\, !\, \ell) \mapsto (M,\, M(\ell))}$$

$$\frac{(M,\, e_1) \mapsto (M',\, e_1')}{(M,\, e_1 := e_2) \mapsto (M',\, e_1' := e_2)} \qquad \frac{(M,\, e_2) \mapsto (M',\, e_2')}{(M,\, v_1 := e_2) \mapsto (M',\, v_1 := e_2')} \qquad \frac{\ell \in dom(M)}{(M,\, \ell := v) \mapsto (M\,[\ell := v],\, \mathbf{0})}$$

$$\frac{(M,\, e) \mapsto (M',\, e')}{(M,\, e[\,]) \mapsto (M',\, e'[\,])} \qquad \frac{}{(M,\, (\Lambda.e)[\,]) \mapsto (M,\, e)} \qquad \frac{(M,\, e) \mapsto (M',\, e')}{(M,\, \mathtt{pack}\, e) \mapsto (M',\, \mathtt{pack}\, e')}$$

$$\frac{(M,\, e_1) \mapsto (M',\, e_1')}{(M,\, \mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2) \mapsto (M',\, \mathtt{open}\, e_1'\, \mathtt{as}\, x\, \mathtt{in}\, e_2)} \qquad \frac{}{(M,\, \mathtt{open}\, (\mathtt{pack}\, v)\, \mathtt{as}\, x\, \mathtt{in}\, e_2) \mapsto (M,\, e_2[v/x])}$$

Figure 1. Small-step operational semantics

state of the abstract machine is described by a pair (M, e) of a memory and an expression. We write $(M, e) \mapsto^j (M', e')$ to mean that there exists a chain of j steps of the form $(M, e) \mapsto (M_1, e_1) \mapsto \ldots \mapsto (M_j, e_j)$ where M_j is M' and e_j is e'. We write $(M, e) \mapsto^* (M', e')$ if $(M, e) \mapsto^j (M', e')$ for some $j \geq 0$. A state (M, e) is *irreducible* if it has no successor in the step relation, that is irred(M, e) if e is a value or (M, e) is a "stuck" state such as $(M, \mathbf{0}(e'))$ or $(M, !\,\ell)$ where $\ell \notin dom(M)$.

We say that (M, e) is safe for k steps if for any reduction $(M, e) \mapsto^j (M', e')$ of $j < k$ steps, either e' is a value or $(M', e') \mapsto (M'', e'')$. Note that any state is safe for 0 steps. A state (M, e) is called safe if it is safe for all $k \geq 0$.

Semantics of Types and Typing Rules. We are interested in constructing methods for proving that a given state is safe. In particular, we want to prove the rules (lemmas) of Figure 2 and also that typability implies safety. For simplicity in this presentation, we avoid the use of type variables; instead of writing $\exists\alpha.\tau$ with type variables α in a type expression τ, we write $\exists F$ where F is a function from types to types. In this paper we assume F is somehow expressible in the underlying logic, but Appel, Richards, and Swadi [7] show how best to handle type expressions and variables in our semantic approach.

The semantic approach taken here is based on types as sets rather than type expressions. We say that a *type* is a set τ of tuples of the form $\langle k, \Psi, M, v\rangle$ where k is a non-negative integer, Ψ is a *memory typing*, that is, a mapping from locations to closed types, M is a memory, and v is a value. Informally, $\langle k, \Psi, M, v\rangle \in \tau$ 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 $\tau \to \tau'$ must execute for at least k steps on v before getting to a stuck state.

As mentioned in the previous section, a naive construction of τ as a set of $\langle k, \Psi, M, v\rangle$, with Ψ as a relation on

$$\frac{}{\Gamma \vDash x : \Gamma(x)}\ \text{(var)} \qquad \frac{}{\Gamma \vDash \mathbf{0} : \mathbf{unit}}\ \text{(unit)}$$

$$\frac{\Gamma\,[x := \tau_1] \vDash e : \tau_2}{\Gamma \vDash \lambda x.e : \tau_1 \to \tau_2}\ \text{(abs)}$$

$$\frac{\Gamma \vDash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vDash e_2 : \tau_1}{\Gamma \vDash (e_1\, e_2) : \tau_2}\ \text{(app)}$$

$$\frac{\Gamma \vDash e : \mu F}{\Gamma \vDash e : F(\mu F)}\ \text{(unfold)} \qquad \frac{\Gamma \vDash e : F(\mu F)}{\Gamma \vDash e : \mu F}\ \text{(fold)}$$

$$\frac{\Gamma \vDash e : \tau}{\Gamma \vDash \mathtt{new}(e) : \mathbf{ref}\ \tau}\ \text{(new)} \qquad \frac{\Gamma \vDash e : \mathbf{ref}\ \tau}{\Gamma \vDash !\, e : \tau}\ \text{(deref)}$$

$$\frac{\Gamma \vDash e_1 : \mathbf{ref}\ \tau_2 \qquad \Gamma \vDash e_2 : \tau_2}{\Gamma \vDash e_1 := e_2 : \mathbf{unit}}\ \text{(assign)}$$

$$\frac{\forall\tau.\, \mathsf{type}(\tau) \Rightarrow \Gamma \vDash e : F(\tau)}{\Gamma \vDash \Lambda.e : \forall F}\ \text{(tabs)}$$

$$\frac{\mathsf{type}(\tau) \qquad \Gamma \vDash e : \forall F}{\Gamma \vDash e[\,] : F(\tau)}\ \text{(tapp)}$$

$$\frac{\mathsf{type}(\tau) \qquad \Gamma \vDash e : F(\tau)}{\Gamma \vDash \mathtt{pack}\, e : \exists F}\ \text{(pack)}$$

$$\frac{\Gamma \vDash e_1 : \exists F \qquad \forall\tau.\, \mathsf{type}(\tau) \;\Rightarrow\; \Gamma\,[x := F(\tau)] \vDash e_2 : \tau_2}{\Gamma \vDash \mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2 : \tau_2}\ \text{(open)}$$

Figure 2. Type-checking lemmas

ℓ and τ, can lead to paradoxes. That is, there is the question of exactly what logic we are using, and what are the metalogical types of τ and Ψ. We will explain in Section 5 how to represent our proof in the Calculus of Inductive Constructions, and in Section 6 how we formulate a solution in higher-order logic using Gödelization.

A type is not just any set of tuples $\langle k, \Psi, M, v\rangle$; it must be well behaved in certain ways. To formally define a type

we must first provide some auxiliary definitions.

Definition 1 (Approx)
The k-approximation of a set is the subset of its elements whose index is less than k; we also extend this notion pointwise to memory typings:

$$\begin{array}{lcl} \lfloor \tau \rfloor_k & \equiv & \{\langle j, \Psi, M, v\rangle \mid j < k \ \wedge \ \langle j, \Psi, M, v\rangle \in \tau\} \\ \lfloor \Psi \rfloor_k & \equiv & \{(\ell \mapsto \lfloor \tau \rfloor_k) \mid \Psi(\ell) = \tau\} \end{array}$$

Intuitively, if we intend to run the program for no more than k more steps, then typechecking with $\lfloor \tau \rfloor_k$ is just as safe as typechecking with τ.

Definition 2 (Well-typed Memory)
A memory M is well-typed to approximation k with respect to a memory typing Ψ iff $dom(\Psi) \subseteq dom(M)$, the domain of M is finite, and the contents of each location $\ell \in dom(\Psi)$ has type $\Psi(\ell)$ to approximation k:

$$\begin{array}{lcl} M :_k \Psi & \equiv & dom(\Psi) \subseteq dom(M) \ \wedge \ \mathsf{finite}(dom(M)) \ \wedge \\ & & \forall j < k. \forall \ell \in dom(\Psi).\ \langle j, \lfloor \Psi \rfloor_j, M, M(\ell)\rangle \in \lfloor \Psi \rfloor_k(\ell) \end{array}$$

The domain of M includes all locations already allocated; elements not in $dom(M)$ are available to be allocated by `new` in future computation steps. We permit the memory to have "extra" allocated locations that are not typed by Ψ, but the model would also work if we required $dom(\Psi) = dom(M)$. Also note that all the tuples required to be in $\Psi(\ell)$ have index strictly less than k; this helps avoid circularity.

Definition 3 (Memory Extension)
A valid memory extension *is defined as follows:*

$$\begin{array}{l} (k, \Psi, M) \sqsubseteq (j, \Psi', M') \ \equiv \\ \quad j \leq k \ \wedge \ (\forall \ell \in dom(\Psi).\ \lfloor \Psi' \rfloor_j(\ell) = \lfloor \Psi \rfloor_j(\ell)) \ \wedge \ M' :_j \Psi' \end{array}$$

Memory extension models what may happen to the memory (and memory typing) during zero or more computation steps. The computation step(s) might allocate a new reference, in which case $dom(\Psi')$ will be a strict superset of $dom(\Psi)$. The step(s) might choose to forget some information, in which case j will be strictly less than k and Ψ' may be more approximate than Ψ—but in this case, the program will now be able to run for at most j more steps instead of k more steps. The computation might store a new value at some location ℓ, but in this case the new value must (approximately) obey the typing given by $\Psi'(\ell)$.

Definition 4 (Type)
A type *is a set τ of tuples of the form $\langle k, \Psi, M, v\rangle$ where v is a value, k is a nonnegative integer, Ψ is a memory typing, and M is a memory such that $M :_k \Psi$, and where the set τ is such that if $\langle k, \Psi, M, v\rangle \in \tau$ and $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$ then $\langle j, \Psi', M', v\rangle \in \tau$.*

$$\begin{array}{lcl} \bot & \equiv & \{\} \\ \mathbf{unit} & \equiv & \{\langle k, \Psi, M, \mathbf{0}\rangle \mid M :_k \Psi\} \\ \tau_1 \rightarrow \tau_2 & \equiv & \{\langle k, \Psi, M, \lambda x.e\rangle \mid M :_k \Psi \ \wedge \\ & & \quad \forall j < k. \forall v, \Psi', M'. \\ & & \quad\quad ((k, \Psi, M) \sqsubseteq (j, \Psi', M') \ \wedge \ \langle j, \Psi', M', v\rangle \in \tau_1) \\ & & \quad\quad \Rightarrow \ e[v/x] :_{j,\Psi',M'} \tau_2\} \\ \mu F & \equiv & \{\langle k, \Psi, M, v\rangle \mid \langle k, \Psi, M, v\rangle \in F^{k+1}(\bot)\} \\ \mathbf{ref}\ \tau & \equiv & \{\langle k, \Psi, M, \ell\rangle \mid M :_k \Psi \ \wedge \ \lfloor \Psi \rfloor_k(\ell) = \lfloor \tau \rfloor_k \ \wedge \\ & & \quad \forall j < k.\ \langle j, \lfloor \Psi \rfloor_j, M, M(\ell)\rangle \in \tau\} \\ \forall F & \equiv & \{\langle k, \Psi, M, \Lambda.e\rangle \mid M :_k \Psi \ \wedge \\ & & \quad \forall j, \Psi', M', \tau.\ ((k, \Psi, M) \sqsubseteq (j, \Psi', M') \ \wedge \ \mathsf{type}(\lfloor \tau \rfloor_j)) \\ & & \quad\quad \Rightarrow \ \forall i < j.\ e :_{i, \lfloor \Psi' \rfloor_i, M'} F(\tau)\} \\ \exists F & \equiv & \{\langle k, \Psi, M, \mathtt{pack}\, v\rangle \mid M :_k \Psi \ \wedge \\ & & \quad \exists \tau.\ \mathsf{type}(\lfloor \tau \rfloor_k) \ \wedge \ \forall j < k.\ \langle j, \lfloor \Psi \rfloor_j, M, v\rangle \in F(\tau)\} \end{array}$$

Figure 3. Type definitions

The essential property of a type is that it is closed under memory extension. This will allow us to prove that if $M(\ell)$ has type τ, and (M, e) steps by computation to (M', e'), then $M(\ell)$ still has type τ, even if other locations in M are stored into.

Definition 5 (Expr : Type)
For any closed expression e and type τ we write $e :_{k,\Psi,M} \tau$ if whenever $(M, e) \mapsto^j (M', e')$ for $j < k$ and (M', e') irreducible, then there exists a memory typing Ψ' such that $(k, \Psi, M) \sqsubseteq (k - j, \Psi', M')$ and $\langle k - j, \Psi', M', e'\rangle \in \tau$; that is,

$$\begin{array}{rcl} e :_{k,\Psi,M} \tau & \equiv & \forall j, M', e'.\ (0 \leq j < k \ \wedge \ (M, e) \mapsto^j (M', e') \\ & & \quad \wedge \ \mathsf{irred}(M', e')) \\ & & \Rightarrow \exists \Psi'.\ (k, \Psi, M) \sqsubseteq (k - j, \Psi', M') \\ & & \quad \wedge \ \langle k - j, \Psi', M', e'\rangle \in \tau \end{array}$$

Intuitively, $e :_{k,\Psi,M} \tau$ means that in a state (M, e), e behaves like an element of τ for k steps of computation. Note that if $e :_{k,\Psi,M} \tau$ and $0 \leq j \leq k$ then $e :_{j,\Psi',M} \tau$, for an appropriate Ψ', by the fact that τ is closed under memory extension. Also, for a value v, and $k > 0$, the statements $v :_{k,\Psi,M} \tau$ and $\langle k, \Psi, M, v\rangle \in \tau$ are equivalent.

We now define the types and type constructors of our language as sets and functions from sets to sets. The definitions appear in Figure 3. The idea (as in the Appel-McAllester indexed model) is that for a value $\lambda x.e$ in $\tau_1 \rightarrow \tau_2$ to be safe for k steps, it must be that if we use up one step by beta-reduction, the resulting expression must be safe for $j < k$ steps.

What's new here is the definition of the quantified types. For a value $\Lambda.e$ to belong (with approximation k) to $\forall F$, we can use only the ith level of the memory-typing Ψ, for i strictly less than k. This approximation is justified, because then we are careful to use only the approximate typing judgment that e is in the ith approximation of $F(\tau)$.

Other types, such as cartesian products, integers, unions, intersections, and so on, are straightforward to define in this manner but we omit them to simplify the presentation.

Open expressions. Up to now we have dealt with closed expressions, as these are the ones that "step" at "run time." Now we turn to expressions with free variables, upon which the static type-checking rules must operate.

Definition 6 (Semantics of Judgment)
A context *is a mapping from lambda calculus variables to types. A* substitution *is a mapping from lambda calculus variables to values. For any context* Γ *and substitution* σ *we write* $\sigma :_{k,\Psi,M} \Gamma$ *("*σ *approximately obeys* Γ*") if for all variables* $x \in dom(\Gamma)$ *we have* $\sigma(x) :_{k,\Psi,M} \Gamma(x)$*.*

Finally, we write $\Gamma \models_k e : \tau$ *to mean that every free variable of* e *is mapped by* Γ *and*

$$\forall \sigma, \Psi, M. (M :_k \Psi \wedge \sigma :_{k,\Psi,M} \Gamma) \Rightarrow \sigma(e) :_{k,\Psi,M} \tau$$

where $\sigma(e)$ *is the result of replacing the free variables in* e *with their values under* σ*.*

That is, the meaning of the judgment $\Gamma \models_k e : \tau$ on an open expression e and type τ can be obtained from our semantics of a similar judgment on closed expressions, so long as we quantify over all (approximately) legitimate substitutions of values for variables.

Definition 7
We write $\Gamma \models e : \tau$ *if for all* $k \geq 0$ *we have* $\Gamma \models_k e : \tau$*. We write* $\models e : \tau$ *to mean* $\Gamma_0 \models e : \tau$ *for the empty context* Γ_0*.*

Note that $\Gamma \models e : \tau$ can be viewed as a three place relation that holds on the context Γ, the term e, and the type τ. Each of the type inference lemmas in Figure 2 states that if certain instances of the relation $\Gamma \models e : \tau$ hold, then certain other instances hold. 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 $\Gamma \models e : \tau$. We now observe that the definitions given above imply the following.

Theorem 8
If $\models e : \tau$*,* τ *is a type, and* M *is a finite memory then* (M, e) *is safe.*

Proof: In a conventional syntactic type theory, the safety theorem (typability implies safety) is difficult (or at least tedious) to prove. Here it follows directly from the definitions. We need to show that for any k, it is safe to execute (M, e) for k steps. From $\models e : \tau$ we have $\Gamma_0 \models_k e : \tau$, and therefore e is closed (since Γ_0 is empty). Choose the empty substition σ_0 and the empty memory typing Ψ_0, and by the definition of $\models_k$ we have

$$M :_k \Psi_0 \wedge \sigma_0 :_{k,\Psi_0,M} \Gamma_0 \Rightarrow \sigma_0(e) :_{k,\Psi_0,M} \tau$$

The two premises are trivially satisfied; applying the trivial substitution we obtain $e :_{k,\Psi_0,M} \tau$. By definition, this means that if (M, e) steps in fewer than k steps to (M', e'), then either e' is a value or another step is possible. □

A "program" is a closed expression that does not contain any location symbols ℓ. When a program begins executing, it steps (by means of `new`) to expressions that may contain location symbols. A conventional subject-reduction proof requires the static type system to be able to type-check programs during execution, so there must be a way to type-check locations ℓ. However, our judgment $\Gamma \models e : \tau$ has no provision to type-check labels (e.g., there is no label-environment to the left of the $\models$ symbol). We don't need to type-check executing programs (since we're not doing subject reduction), and so we don't need to type-check location symbols. This is one reason that, in the proof of Theorem 8, we are able to choose the empty memory-typing Ψ_0.

3 Proofs of Types

In order to prove that a program e with type τ is safe to execute (Theorem 8), it must be the case that τ is a type. Next, we prove that each of the type constructors shown in Figure 3 is a type, or produces a type when applied to valid arguments. The fact that $\bot$ and $\mathbf{unit}$ are types follows immediately from their definitions. To prove that $\tau_1 \to \tau_2$ is a type, we need the following lemma which says that memory extension is transitive. The proof is given in Appendix A.

Lemma 9 (Memory Extension Transitive)
If $(k_1, \Psi_1, M_1) \sqsubseteq (k_2, \Psi_2, M_2)$ *and* $(k_2, \Psi_2, M_2) \sqsubseteq (k_3, \Psi_3, M_3)$ *then* $(k_1, \Psi_1, M_1) \sqsubseteq (k_3, \Psi_3, M_3)$*.*

Lemma 10 (Type $\tau_1 \to \tau_2$)
If τ_1 *and* τ_2 *are types then* $\tau_1 \to \tau_2$ *is also a type.*

Proof: First, if $\langle k, \Psi, M, v \rangle \in \tau_1 \to \tau_2$ then $M :_k \Psi$. This is immediate from the definition of $\to$.

Next, we must prove that $\tau_1 \to \tau_2$ is closed under valid memory extension. Suppose that $\langle k, \Psi, M, v \rangle \in \tau_1 \to \tau_2$ and $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. Note that $\langle k, \Psi, M, v \rangle \in \tau_1 \to \tau_2$ implies that v is of the form $\lambda x.e$. We must prove that $\langle j, \Psi', M', \lambda x.e \rangle \in \tau_1 \to \tau_2$. We first require that $M' :_j \Psi'$ which immediately follows from $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. Next, let $(j, \Psi', M') \sqsubseteq (i, \Psi'', M'')$ and $\langle i, \Psi'', M'', v'' \rangle \in \tau_1$ for $i < j$ and some value v'' — we have to show $e[v''/x] :_{i,\Psi'',M''} \tau_2$. Since $i < j$ and, by the definition of $\sqsubseteq$, we have $j \leq k$, it follows that $i < k$; by Lemma 9 we have $(k, \Psi, M) \sqsubseteq (i, \Psi'', M'')$; and we already have $\langle i, \Psi'', M'', v'' \rangle \in \tau_1$. These three statements together with $\langle k, \Psi, M, \lambda x.e \rangle \in \tau_1 \to \tau_2$ and the definition of $\to$ allow us to conclude that $e[v''/x] :_{i,\Psi'',M''} \tau_2$. □

Next, we prove that $\mathbf{ref}\ \tau$ and $\forall F$ are types, given appropriate τ and F, respectively. The proofs for the remaining type constructors are given in Appendix A.

Lemma 11 (Type ref)
If τ is a type then $\mathbf{ref}\ \tau$ is a type.

Proof: We have to show that if $\langle k, \Psi, M, v\rangle \in \mathbf{ref}\ \tau$ then $M :_k \Psi$. But this is immediate from the definition of $\mathbf{ref}$.

To show that $\mathbf{ref}\ \tau$ is closed under memory extension, suppose that $\langle k, \Psi, M, v\rangle \in \mathbf{ref}\ \tau$ and $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. We must prove that $\langle j, \Psi', M', v\rangle \in \mathbf{ref}\ \tau$. There are three parts to the proof.

First, we must show that $M' :_j \Psi'$, but this follows directly from $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$.

Second, by the definition of $\mathbf{ref}$, since $\langle k, \Psi, M, v\rangle \in \mathbf{ref}\ \tau$, it follows that v is some location ℓ and $\lfloor\Psi\rfloor_k(\ell) = \lfloor\tau\rfloor_k$. The latter implies that $\ell \in dom(\Psi)$. Then, from $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$ it follows that $\lfloor\Psi'\rfloor_j(\ell) = \lfloor\Psi\rfloor_j(\ell)$, and hence, $\ell \in dom(\Psi')$. In addition, from $\lfloor\Psi\rfloor_k(\ell) = \lfloor\tau\rfloor_k$ and $j \leq k$, it follows by definition 1 (Approx) that $\lfloor\Psi\rfloor_j(\ell) = \lfloor\tau\rfloor_j$, and so we may conclude that $\lfloor\Psi'\rfloor_j(\ell) = \lfloor\tau\rfloor_j$ by transitivity of set equality.

For the third part of the proof, let $i < j$; we must show that $\langle i, \lfloor\Psi'\rfloor_i, M', M'(\ell)\rangle \in \tau$. We concluded above that $M' :_j \Psi'$ and that $\ell \in dom(\Psi')$. From $M' :_j \Psi'$, using the fact that $\ell \in dom(\Psi')$ and that $i < j$, we may conclude that $\langle i, \lfloor\Psi'\rfloor_i, M', M'(\ell)\rangle \in \lfloor\Psi'\rfloor_j(\ell)$. Finally, from $\langle i, \lfloor\Psi'\rfloor_i, M', M'(\ell)\rangle \in \lfloor\Psi'\rfloor_j(\ell)$ and $\lfloor\Psi'\rfloor_j(\ell) = \lfloor\tau\rfloor_j$, since $i < j$, it follows that $\langle i, \lfloor\Psi'\rfloor_i, M', M'(\ell)\rangle \in \tau$. □

Lemma 12 (Type $\forall F$)
If F is a function from types to types then $\forall F$ is a type.

Proof: First, if $\langle k, \Psi, M, v\rangle \in \forall F$ then $M :_k \Psi$. This is immediate from the definition of $\forall F$.

Next, to prove that $\forall F$ is closed under memory extension, suppose that $\langle k, \Psi, M, v\rangle \in \forall F$ and $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. Note that $\langle k, \Psi, M, v\rangle \in \forall F$ implies that v is of the form $\Lambda.e$. We must show that $\langle j, \Psi', M', \Lambda.e\rangle \in \forall F$. The proof has two parts. First, we need to prove that $M' :_j \Psi'$ but this is immediate from $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. For the second part of the proof, let $(j, \Psi', M') \sqsubseteq (i, \Psi'', M'')$ and $\mathsf{type}(\lfloor\tau\rfloor_i)$ for some set τ — we must show that $e :_{i', \lfloor\Psi''\rfloor_{i'}, M''} F(\tau)$ for all $i' < i$. Since memory extension is transitive (Lemma 9), we have that $(k, \Psi, M) \sqsubseteq (i, \Psi'', M'')$. Now, from $\langle k, \Psi, M, \Lambda.e\rangle \in \forall F$ we may conclude that for any $i' < i$, $e :_{i', \lfloor\Psi''\rfloor_{i'}, M''} F(\tau)$. □

4 Proofs of the Typing Lemmas

We now prove each of the type inference lemmas in Figure 2. There is a type inference lemma for each case in the grammar of lambda terms (except for locations ℓ) 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 type inference lemma for $\mathbf{0}$ stating $\Gamma \models 0 : \mathbf{unit}$ follows directly from the definition of $\mathbf{unit}$.

In this section we prove the type theorems for type abstraction and type application. Proofs for the remaining type-checking rules in Figure 2 are given in Appendix B (Theorems 32–42) .

Lemma 13 (Closed Type Application)
If e is a closed term, τ is a type, and F is a function from types to types such that $e :_{k,\Psi,M} \forall F$, then $e[\,] :_{k,\Psi,M} F(\tau)$.

Proof: We must prove $e[\,] :_{k,\Psi,M} F(\tau)$ under the premises of the lemma. Since F is a function from types to types, by Lemma 12 $\forall F$ is a type. Since $e :_{k,\Psi,M} \forall F$ we have that (M, e) is safe for k steps and if (M, e) reduces to (M', v) (where v is a value) in fewer than k steps, then v must be of the form $\Lambda.e'$. Hence, the state $(M, e[\,])$ either reduces for k steps without reaching a state of the form $(M', v[\,])$ or there exists e' and M' such that $(M, e[\,]) \mapsto^j (M', (\Lambda.e')[\,])$ with $j < k$. In the first case we have that $(M, e[\,])$ is safe for k steps and (M, e) does not reduce to a value in fewer than k steps and hence $e[\,] :_{k,\Psi,M} F(\tau)$. In the second case, it follows from the operational semantics in Figure 1 that $(M, e) \mapsto^j (M', \Lambda.e')$. Since $e :_{k,\Psi,M} \forall F$, definition 5 (Expr : Type) implies that there exists a memory typing Ψ' such that $(k, \Psi, M) \sqsubseteq (k - j, \Psi', M')$ and $\langle k - j, \Psi', M', \Lambda.e'\rangle \in \forall F$.

Next, pick the memory typing $\lfloor\Psi'\rfloor_{k-j-1}$. Then, the following information-forgetting memory extension holds: $(k - j, \Psi', M') \sqsubseteq (k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M')$. From $\mathsf{type}(\tau)$ it follows by definitions 4 and 1 (Type and Approx) that $\mathsf{type}(\lfloor\tau\rfloor_{k-j-1})$. Then from $\Lambda.e' :_{k-j,\Psi',M'} \forall F$ and the definition of $\forall F$ we may conclude that for any $i < (k - j - 1)$, $e' :_{i, \lfloor\lfloor\Psi'\rfloor_{k-j-1}\rfloor_i, M'} F(\tau)$ — that is, $e' :_{i, \lfloor\Psi'\rfloor_i, M'} F(\tau)$ (from $i < (k - j - 1)$ and definition 1 (Approx)).

Since τ is a type, $F(\tau)$ is a type. Also, the following information-forgetting memory extension holds: $(k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M') \sqsubseteq (i, \lfloor\Psi'\rfloor_i, M')$. Then, first, by Lemma 9 we have $(k, \Psi, M) \sqsubseteq (i, \lfloor\Psi'\rfloor_i, M')$. Second, since $(M'.(\Lambda.e')[\,]) \mapsto (M', e')$, we have $(M, e[\,]) \mapsto^{j+1} (M', e')$. Third, we have $e' :_{i, \lfloor\Psi'\rfloor_i, M'} F(\tau)$ where $\mathsf{type}(F(\tau))$. These three statements imply (since they hold for any $i < (k-(j+1))$) that $e[\,] :_{k,\Psi,M} F(\tau)$. □

Theorem 14 (Type Application)
Let Γ be a context, let F be a function from types to types, and let τ be a type. If $\Gamma \models e : \forall F$ then $\Gamma \models e[\,] : F(\tau)$.

Proof: We must prove that for any $k \geq 0$ we have $\Gamma \models_k e[\,] : F(\tau)$. More specifically, for any M, Ψ, and σ such that $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$, we must show $\sigma(e[\,]) :_{k,\Psi,M} F(\tau)$. Suppose $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$. By the premise of the theorem we have $\sigma(e) :_{k,\Psi,M} \forall F$ and $\mathsf{type}(\tau)$. The result now follows from Lemma 38. □

Before we can prove the typing rules for type abstraction, we must define the notion of a nonexpansive functional.

Definition 15
A nonexpansive functional *is a function F from types to types such that for any type τ and $k \geq 0$ we have*

$$\lfloor F(\tau) \rfloor_k = \lfloor F(\lfloor \tau \rfloor_k) \rfloor_k$$

The term "nonexpansive" is explained in more detail by Appel and McAllester, who show that all the functionals that can be built by compositions of our type constructors are nonexpansive.

Theorem 16 (Type Abstraction)
Let Γ be a context and let F be a nonexpansive type functional. If $\Gamma \models e : F(\tau)$ for any type set τ, then $\Gamma \models \Lambda.e : \forall F$.

Proof: We must show that for any $k \geq 0$ and M, Ψ and σ such that $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$ we have $\sigma(\Lambda.e) :_{k,\Psi,M} \forall F$. Suppose $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$. Let j,v, Ψ', M', and τ be such that $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$ and $\mathsf{type}(\lfloor \tau \rfloor_j)$. By the definition of $\forall$ it suffices to show that for any $i < j$ we have $\sigma(e) :_{i,\lfloor \Psi' \rfloor_i, M'} F(\tau)$. Since $\lfloor \tau \rfloor_j$ is a type, given the premise of theorem it follows that $\Gamma \models e : F(\lfloor \tau \rfloor_j)$. Hence $\sigma(e) :_{k,\Psi,M} F(\lfloor \tau \rfloor_j)$. Also, since F is a function from types to types, $F(\lfloor \tau \rfloor_j)$ is a type.

The following information-forgetting memory extension holds: $(j, \Psi', M') \sqsubseteq (i, \lfloor \Psi' \rfloor_i, M')$. Then by Lemma 9 it follows that $(k, \Psi, M) \sqsubseteq (i, \lfloor \Psi' \rfloor_i, M')$. Hence, from $\sigma(e) :_{k,\Psi,M} F(\lfloor \tau \rfloor_j)$ and $\mathsf{type}(F(\lfloor \tau \rfloor_j))$ we can conclude that $\sigma(e) :_{i,\lfloor \Psi' \rfloor_i, M'} F(\lfloor \tau \rfloor_j)$. Now since $i < j$, definitions 1, 4, and 5 (Approx, Type, and Expr : Type) allow us to conclude that $\sigma(e) :_{i,\lfloor \Psi' \rfloor_i, M'} \lfloor F(\lfloor \tau \rfloor_j) \rfloor_j$. Using the premise that F is nonexpansive we have that $\sigma(e) :_{i,\lfloor \Psi' \rfloor_i, M'} \lfloor F(\tau) \rfloor_j$. But since $i < j$, definition 1 (Approx) implies that $\sigma(e) :_{i,\lfloor \Psi' \rfloor_i, M'} F(\tau)$. □

5 Representation in CiC

At the beginning of Section 2, we informally defined types as collections of tuples $\langle k, \Psi, M, v \rangle$, where in particular Ψ is a mapping from location to types. This informal definition characterizes a proper class rather than a set, since for example it allows pathological cases like:

$$\langle k, \Psi, M, v \rangle \in \tau \text{ and } \Psi(\ell) = \tau$$

Any attempt toward a set-theoretic formalization starting from this definition is therefore doomed to failure.

What makes such a formalization possible is that all definitions given in Section 2 obey the following invariant: when considering a tuple $\langle k, \Psi, M, v \rangle$ we do not require Ψ to be defined beyond $\lfloor \Psi \rfloor_k$. Hence our types do indeed form a set, which can be constructed using recursively defined sets Types_k and $\mathrm{MemTypes}_k$ as follows:

$\tau \in \mathrm{Types}_0$ iff $\tau = \{\}$
$\tau \in \mathrm{Types}_{k+1}$ iff $\forall \langle j, \Psi, M, v \rangle \in \tau.\ j \leq k \wedge \Psi \in \mathrm{MemTypes}_j$
$\Psi \in \mathrm{MemTypes}_k$ iff $\forall \ell \in dom(\Psi).\ \Psi(\ell) \in \mathrm{Types}_k$
$\tau \in \mathrm{Types}$ iff $\forall k.\ \lfloor \tau \rfloor_k \in \mathrm{Types}_k$.

The Calculus of Inductive Constructions, upon which the Coq system is based, can represent the above definitions quite directly:

```
Fixpoint itype [k : nat] : Type :=
  Cases k
    of O => UnitT
     | (S k') =>
        (prodT (itype k')
          ((location -> (itype k'))
            -> memory -> exp -> Prop))
  end.

Definition imemtype [k : nat] : Type :=
  location -> (itype k).

Definition type: Type := (k: nat) (itype k).
```

Each set Types_k is modeled using a product type `(itype k)` in CiC. More specifically, Types_0 is modeled by the unit type `UnitT`, while Types_{k+1} is given by the product of the representation of Types_k and the set of membership functions for triples $\langle \Psi, M, v \rangle$, where Ψ has type `(imemtype k)`.

Given an object `tau` of type `type`, its k-th approximation will correspond to the application `(tau k)`. To check that a 4-tuple $\langle k, \Psi, M, v \rangle$ is in `tau`, we will need to apply `tau` to $k + 1$, and take the second component:

```
(sndT (tau (S k)) Psi M v) : Prop
```

The definition of the types and type constructors presented in Figure 3 can be given by recursion on the natural numbers. At each step, we have to construct an object of type `(itype k)`. The case $k = 0$ is trivial, since `UnitT` is the only object belonging to `(itype 0)`. For the case $k + 1$, we only have to provide a formula which decides which tuples $\langle k, \Psi, M, v \rangle$ will belong to the type.

Figure 4 illustrates the representation for the type constructor for mutable references. The predicate

```
Definition refTy [tau : type] : type :=
  (nat_rect itype IT
    ([k : nat][tauk : (itype k)]
     (pairT tauk
      ([psi: (imemtype k)][m: memory][v: exp]
       (Ex [l : location]
       (Ex [v' : exp]
           (v = (loc l))
        /\ ((m l) = (Some exp v'))
        /\ (imemtype_sat k psi m)
        /\ (eqT (itype k) (psi l) (tau k))
        /\ (All [j : nat]
             (h : (lt j k))
             ((sndT (tau (S j)))
              (imemtype_approx k j h psi)
              m v')))))))).
```

Figure 4. Definition of ref in CiC

```
imemtype_sat : (k:nat)
               (imemtype k) -> memory
                            -> Prop
```

models the relation $M :_k \Psi$, while we use the function

```
imemtype_approx : (k,j:nat)
                  (lt j k) -> (imemtype k)
                           -> (imemtype j)
```

to "lower" an approximation to the correct index. Both of these are straightforwardly defined in Coq.

6 Representation in H.O.L.

Higher-order logic does not provide as convenient a mechanism for making stratified metalogical types; we must construct the stratification ourselves. We do so by constructing a Gödelization of type expressions.

A naive Gödelization would proceed as follows. We want a relation ρ between the natural numbers and all the type expressions that can be constructed from the operators in Figure 3. Unfortunately, the definitions of some of those operators would refer to ρ, leading to a circularity. We resolve this circularity by constructing a hierarchy of relations; the semantics of the types in one level of the hierarchy can make use of lower levels. We now present the metalogical types, in higher-order logic, of what we will construct:

$$\begin{array}{lcl} \mathit{Exp} & = & \textit{the type of lambda-expressions} \\ \mathit{Loc} & = & \mathit{Nat} \\ \mathit{Mem} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Exp} \\ \mathit{Term} & = & \mathit{Nat} \\ \mathit{MemType} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Term} \\ \mathit{Type} & = & \mathit{Nat} \times \mathit{MemType} \times \mathit{Mem} \times \mathit{Exp} \to o \\ \mathit{Rep} & = & \mathit{Term} \to \mathit{Type} \end{array}$$

$Exp\ e$ is a lambda-expression; $Loc\ \ell$ is an addressable memory location; $Mem\ M$ is a memory; $Term\ t$ is a Gödel number; $MemType\ \Psi$ is a memory typing, but in this model it maps locations to terms instead of types; $Type\ \tau$ is a set of 4-tuples, as before; $Rep\ \rho$ is a representation function (Gödel numbering).

Definition 17 (Approx)
We define $\lfloor\tau\rfloor_k$ as before. However, since now memory typing maps locations into terms, we need to parametrize the approximation of a memory typing with respect to a representation function:

$$\lfloor\Psi\rfloor_{\rho,k} \equiv \{(\ell \mapsto \lfloor\rho(t)\rfloor_k) |\ \Psi(\ell) = t\}$$

Note that the metalogical type of $\lfloor\Psi\rfloor_{\rho,k}$ is $Loc \to Type$, which is different than the metalogical type of Ψ.

All the definitions given in Section 2 follow through, but need to be parametrized by ρ as well.

Definition 18 (Well-typed Memory)

$$\begin{array}{l} M :_{\rho,k} \Psi \ \equiv \\ \quad dom(\Psi) \subseteq dom(M) \ \wedge\ \mathsf{finite}(dom(M)) \ \wedge \\ \quad \forall j < k. \forall \ell \in dom(\Psi). \\ \qquad \exists \Psi'.\, \lfloor\Psi'\rfloor_{\rho,j} = \lfloor\Psi\rfloor_{\rho,j} \ \wedge\ \langle j, \Psi', M, M(\ell)\rangle \in \lfloor\Psi\rfloor_{\rho,j}(\ell) \end{array}$$

Definition 19 (Memory Extension)

$$\begin{array}{l} (k, \Psi, M) \sqsubseteq_\rho (j, \Psi', M') \ \equiv \\ \quad j \le k \ \wedge\ (\forall \ell \in dom(\Psi).\, \lfloor\Psi'\rfloor_{\rho,j}(\ell) = \lfloor\Psi\rfloor_{\rho,j}(\ell)) \ \wedge\ M' :_{\rho,j} \Psi' \end{array}$$

Definition 20 (Type)
We say $\mathsf{type}_\rho(\tau)$ if whenever $\langle k, \Psi, M, v\rangle \in \tau$ we have $M :_{\rho,k} \Psi$ and if $(k, \Psi, M) \sqsubseteq_\rho (j, \Psi', M')$ then $\langle j, \Psi', M', v\rangle \in \tau$.

Definition 21 (Expr : Type)

$$\begin{array}{rcl} e :_{\rho,k,\Psi,M} \tau & \equiv & \forall j, M', e'.\, (0 \le j < k \ \wedge\ (M, e) \mapsto^j (M', e') \\ & & \qquad \wedge\ \mathsf{irred}(M', e')) \\ & & \Rightarrow \exists \Psi'.\, (k, \Psi, M) \sqsubseteq_\rho (k - j, \Psi', M') \\ & & \qquad \wedge\ \langle k - j, \Psi', M', e'\rangle \in \tau \end{array}$$

We define *pretype* constructors, from which we will later define type constructors. Each pretype constructor (written with an $\overline{overbar}$) needs a ρ parameter.

$$\begin{array}{rcl} \overline{\bot}_\rho & \equiv & \{\} \\ \overline{\mathbf{unit}}_\rho & \equiv & \{\langle k, \Psi, M, \mathbf{0}\rangle \mid M :_{\rho,k} \Psi\} \\ \tau_1 \overline{\to}_\rho \tau_2 & \equiv & \{\langle k, \Psi, M, \lambda x.e\rangle \mid M :_{\rho,k} \Psi \ \wedge \\ & & \quad \forall j < k. \forall v, \Psi', M'. \\ & & \qquad ((k, \Psi, M) \sqsubseteq_\rho (j, \Psi', M') \ \wedge\ \langle j, \Psi', M', v\rangle \in \tau_1) \\ & & \qquad \Rightarrow\ e[v/x] :_{\rho,j,\Psi',M'} \tau_2\} \\ \overline{\mu}_\rho F & \equiv & \{\langle k, \Psi, M, v\rangle \mid \langle k, \Psi, M, v\rangle \in F^{k+1}(\bot\rho)\} \\ \overline{\mathbf{ref}}_\rho \tau & \equiv & \{\langle k, \Psi, M, \ell\rangle \mid M :_{\rho,k} \Psi \ \wedge\ \lfloor\Psi\rfloor_{\rho,k}(\ell) = \lfloor\tau\rfloor_k \wedge \\ & & \quad \forall j < k.\ \langle j, \Psi, M, M(\ell)\rangle \in \tau\} \\ \overline{\forall}_\rho F & \equiv & \{\langle k, \Psi, M, \Lambda.e\rangle \mid M :_{\rho,k} \Psi \ \wedge \\ & & \quad \forall j, \Psi', M', \tau.\, ((k, \Psi, M) \sqsubseteq_\rho (j, \Psi', M') \ \wedge\ \mathsf{type}_\rho(\lfloor\tau\rfloor_j)) \\ & & \qquad \Rightarrow\ \forall i < j.\, e :_{\rho,i,\Psi',M'} F(\tau)\} \\ \overline{\exists}_\rho F & \equiv & \{\langle k, \Psi, M, \mathsf{pack}\, v\rangle \mid M :_{\rho,k} \Psi \ \wedge \\ & & \quad \exists \tau.\, \mathsf{type}_\rho(\lfloor\tau\rfloor_k) \ \wedge\ \forall j < k.\ \langle j, \Psi, M, v\rangle \in F(\tau)\} \end{array}$$

Now, having defined the pretype constructors, we are free to Gödelize them. Level 0 of the hierarchy is a relation that maps every Gödel number to the bottom type, that is, $\mathsf{rep}_0(t) = \{\}$. Level $i+1$ of the hierarchy is defined as follows:

$$\begin{array}{l}
\mathsf{rep}_{i+1}(\langle 0,0\rangle) = \overline{\bot}_{\mathsf{rep}_i} \\
\mathsf{rep}_{i+1}(\langle 2,0\rangle) = \overline{\mathbf{unit}}_{\mathsf{rep}_i} \\
\mathsf{rep}_{i+1}(\langle 3,\langle t_1,t_2\rangle\rangle) = \mathsf{rep}_{i+1}(t_1)\overline{\Rightarrow}_{\mathsf{rep}_i}\mathsf{rep}_{i+1}(t_2) \\
\ldots \\
\mathsf{rep}_{i+1}(\langle 5,t\rangle) = \overline{\mathbf{ref}}_{\mathsf{rep}_i}(\mathsf{rep}_{i+1}(t)) \\
\ldots
\end{array}$$

We use the notation $\langle i,j\rangle$ for the injective mapping from pairs of natural numbers to the natural numbers. We don't show here the representation of $\overline{\mu}F$, $\overline{\forall}F$ and $\overline{\exists}F$ because this would require a Gödelization of type-functions as well as types. The way we handle this in the proof of a full-scale type system is to Gödelize type expressions (with free deBruijn variables) instead of types.

Definition 22 (Type constructors)

$$\begin{array}{llll}
\bot & = \bigcup_k \lfloor \overline{\bot}_{\mathsf{rep}_k} \rfloor_k & \mathbf{unit} & = \bigcup_k \lfloor \overline{\mathbf{unit}}_{\mathsf{rep}_k} \rfloor_k \\
\tau_1 \to \tau_2 & = \bigcup_k \lfloor \tau_1 \overline{\Rightarrow}_{\mathsf{rep}_k} \tau_2 \rfloor_k & & \\
\mu F & = \bigcup_k \lfloor \overline{\mu}_{\mathsf{rep}_k} F \rfloor_k & \mathbf{ref}\ \tau & = \bigcup_k \lfloor \overline{\mathbf{ref}}_{\mathsf{rep}_k} \tau \rfloor_k \\
\forall F & = \bigcup_k \lfloor \overline{\forall}_{\mathsf{rep}_k} F \rfloor_k & \exists F & = \bigcup_k \lfloor \overline{\exists}_{\mathsf{rep}_k} F \rfloor_k
\end{array}$$

Definition 23 (Semantics of Judgment)
For any value context Γ and value substitution σ we write $\sigma :_{\rho,k,\Psi,M} \Gamma$ ("σ approximately obeys Γ") if for all variables $x \in dom(\Gamma)$ we have $\sigma(x) :_{\rho,k,\Psi,M} \Gamma(x)$.

We write $\Gamma \models_k e : \tau$ to mean that every free variable of e is mapped by Γ and

$$\begin{array}{l}
\forall \sigma, \Psi, M.\, (M :_{\mathsf{rep}_k,k} \Psi \ \wedge\ \sigma :_{\mathsf{rep}_k,k,\Psi,M} \Gamma) \\
\qquad \Rightarrow\ \sigma(e) :_{\mathsf{rep}_k,k,\Psi,M} \tau
\end{array}$$

Theorem 24
Using these definitions as the interpretation of the typing operators, all the rules of Figure 2 hold, as well as the statement of Theorem 8 (typability implies safety).

Proof: The proof corresponds closely the the proof shown in Section 4. We are implementing a machine-checked version of this higher-order-logic proof in the Twelf system. That proof is for von Neumann machines instead of for lambda-calculus, since our application is in proof-carrying code for a real machine. □

To better explain the correspondence between our higher-order logic proof and the proof shown in Section 4, we make a few remarks.

All (positive) representation levels are defined on the same set of terms, which constitutes the set of valid terms. Intuitively, each valid term t corresponds bijectively with a type τ freely built using the constructors of Figure 3, and increasing representation levels offer us increasingly better approximations of that type τ. This idea is formalized by the following key result:

Lemma 25
Let t be a term, and τ its corresponding type. For each i we have $\lfloor \tau \rfloor_i = \lfloor \mathsf{rep}_i(t) \rfloor_i$.

Proof: By a nested induction argument. The primary induction is on i, the secondary one on the structure of t. The case $i = 0$ is trivial, since $\lfloor \tau \rfloor_0 = \{\} = \lfloor \mathsf{rep}_0(t) \rfloor_0$. The case for $(i+1)$ is done by case analysis on the structure of t. □

7 Eliminating Noncomputational Steps

Our application for this semantics is in a proof-carrying code system that can provide safety proofs (derived from type-checking) for an ordinary machine-language program, where the machine itself has no notion of types. In λ-calculus terms, we would like a semantics of types that can yield safety proofs for an entirely untyped operational lambda-calculus. For example, we wish to avoid explicit fold and unfold steps in calculating with recursive types, and indeed the type system we have demonstrated is equirecursive (μF is *equal* to $F(\mu F)$) instead of isorecursive (μF isomorphic, via fold/unfold, to $F(\mu F)$).

However, our operational semantics in Figure 1 has explicit "run-time" steps for opening an existential and applying a universal ($\mathtt{open}(\mathtt{pack}\, v)$ and $(\Lambda.e)[]$). A real machine has no such instructions.

We would like to eliminate $e[\,]$ and `open` from our operational calculus and use typing rules like this:

$$\frac{\forall \tau.\, \mathsf{type}(\tau) \Rightarrow \Gamma \models e : F(\tau)}{\Gamma \models e : \forall F}\ \text{(tabs')}$$

$$\frac{\Gamma \models e : \forall F \qquad \mathsf{type}(\tau)}{\Gamma \models e : F(\tau)}\ \text{(tapp')}$$

But there's a minor technical problem. Consider the statement of Lemma 13: If e is a closed term and F is a type functional such that $e :_{k,\Psi,M} \forall F$ then, for any type set τ, $e[\,] :_{k,\Psi,M} F(\tau)$. If we restate this to conclude that $e :_{k,\Psi,M} F(\tau)$, the lemma will not hold; in fact, all we can prove is $e :_{k-1,\Psi,M} F(\tau)$. If we view the index k as counting the number of computation steps that it's safe to execute, then applying a universal uses up one computation step. The reason can be seen in the definition of the $\forall F$ operator (Figure 3); in judging $v :_{k,\Psi,M} \forall F$, we judge whether $v :_{i,\Psi,M} F(\tau)$ where i is strictly less than k. We have no information about whether $v :_{k,\Psi,M} F(\tau)$.

It is for this reason that we put explicit computation steps for applying a universal and for unpacking an existential into our operational calculus: it makes the proof simpler.

But in our prototype proof-carrying code system, these extra computational steps are ugly: they require the compiler to generate a no-op instruction each time it unpacks an existential. The purpose of this no-op is to use up one computational step to satisfy the proof.

To eliminate this noncomputational step from our lambda-calculus model, we can simply change the statement of our lemma to read: If $\forall k.(e :_{k,\Psi,M} \forall F)$ then $\forall k.(e :_{k,\Psi,M} F(\tau))$. This weaker lemma will be enough to prove the typing judgment (tapp') shown above.

However, the way our model of types for von Neumann machines is structured, this weaker lemma is not as useful. When we reason about recursive functions, we want to prove that "if this computation is safe for k steps, then it's safe for $k+1$ steps;" in such a proof, we can't easily assume that e is safe for arbitrary k.

We will sketch a solution. We will define a step relation that allows real steps (from the specification of the real machine) and artificial steps (type application and `open`); we ensure that at most a bounded number of artificial steps can be taken between real steps (by decrementing an artificial counter in the state). For any program e, there is a number N such that e never executes more than N consecutive artificial steps. In our model, we will define a typing judgement $e :_{N,k,\Psi,M} \tau$ that means, e executes no more than N consecutive artificial steps before the first real step, and then executes no more than N artificial steps between real steps; and is safe for k real steps.

8 Conclusion

The indexed model of types was derived by considering the notion of approximations inherent in domain theory. But the particular advantage of the indexed model is that it permits simple and direct proofs, without the need to "import" large mathematical theories, such as domain theory or category theory.

We have successfully adapted the indexed model to the difficult task of modeling impredicative-polymorphic mutable references. We continue to be guided by the domain-theoretic idea of approximating everything in sight. The resulting proofs, though more complicated than those for a model without references, are still short enough to permit implementation as machine-checked proofs in a simple higher-order logic, or in the calculus of inductive constructions.

References

[1] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 334–344, Los Alamitos, California, 1998. IEEE Computer Society Press.

[2] A. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. In *17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86. IEEE, July 2002.

[3] A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, June 2001.

[4] A. W. Appel and A. 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, Jan. 2000.

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

[6] A. W. Appel, N. Michael, A. Stump, and R. Virga. A trustworthy proof checker. In I. Cervesato, editor, *Foundations of Computer Security workshop*, pages 37–48. DIKU, July 2002. diku.dk/publikationer/tekniske.rapporter/2002/02-12.pdf.

[7] A. W. Appel, C. D. Richards, and K. N. Swadi. A kind system for typed machine language. Available at `http:// www.cs.princeton.edu/~appel/papers/kinding.pdf`, Oct. 2002.

[8] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. *submitted for publication*, 2002.

[9] T. Coquand, C. A. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81:123–167, 1989.

[10] K. Crary. Toward a foundational typed assembly language. In *POPL '03: 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 2003.

[11] M. P. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. Technical Report CSR-96-2, School of Computer Science, The University of Birmingham, 1996. 30pp., available from `http://www.cs.bham.ac.uk/`.

[12] R. Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, 1996.

[13] P. B. Levy. Possible world semantics for general storage in call-by-value. In *Computer Science Logic, 16th International Workshop, CSL 2002 Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 232–246, Edinburgh, Scotland, UK, Sept. 2002. Springer.

[14] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, Jan. 1996.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

[16] C. Schürmann. *Automating the Meta-Theory of Deductive Systems*. Ph. D. thesis, Carnegie Mellon University, Pittsburgh, PA, 2000.

[17] R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.

A Proofs of Types

Lemma 9 (Memory Extension Transitive)
If $(k_1, \Psi_1, M_1) \sqsubseteq (k_2, \Psi_2, M_2)$ *and* $(k_2, \Psi_2, M_2) \sqsubseteq (k_3, \Psi_3, M_3)$ *then* $(k_1, \Psi_1, M_1) \sqsubseteq (k_3, \Psi_3, M_3)$.

Proof: The proof is in three parts. First, from the premises of the lemma and the definition of $\sqsubseteq$ we have $k_2 \leq k_1$ and $k_3 \leq k_2$. It follows that $k_3 \leq k_1$. Second, suppose $\ell \in dom(\Psi_1)$. Then the first premise of the lemma allows us to conclude that $\lfloor \Psi_2 \rfloor_{k_2}(\ell) = \lfloor \Psi_1 \rfloor_{k_2}(\ell)$. It follows that $\ell \in dom(\Psi_2)$. Then, from the second premise of the lemma we have $\lfloor \Psi_3 \rfloor_{k_3}(\ell) = \lfloor \Psi_2 \rfloor_{k_3}(\ell)$. From $\lfloor \Psi_2 \rfloor_{k_2}(\ell) = \lfloor \Psi_1 \rfloor_{k_2}(\ell)$ and $k_3 \leq k_2$ it follows that $\lfloor \Psi_2 \rfloor_{k_3}(\ell) = \lfloor \Psi_1 \rfloor_{k_3}(\ell)$ (by definition 1 (Approx)). Hence we may conclude that $\lfloor \Psi_3 \rfloor_{k_3}(\ell) = \lfloor \Psi_1 \rfloor_{k_3}(\ell)$ by transitivity of set equality. For the third part of our proof we must show that $M_3 :_{k_3} \Psi_3$ but this is immediate from the second premise of the lemma. □

Lemma 10 (Type $\tau_1 \to \tau_2$)
If τ_1 *and* τ_2 *are types then* $\tau_1 \to \tau_2$ *is also a type.*

Proof: Given in Section 3. □

Before we can prove that μF is a type, we must define the notion of a well-founded functional.

Definition 26 (Well Founded)
A well founded functional *is a function* F *from types to types such that for any type* τ *and* $k \geq 0$ *we have*

$$\lfloor F(\tau) \rfloor_{k+1} = \lfloor F(\lfloor \tau \rfloor_k) \rfloor_{k+1}$$

Appel and McAllester [5] show that type functions $\mathbf{ref}$ and $\to$ are well founded, and that the composition of well founded and nonexpansive functionals (in any order) is well founded. Note that if F is a function from types to types and τ is a type then $F^k(\tau)$ is a type for any $k \geq 0$.

Lemma 27
For F *well founded and* $j \leq k$, *for any* τ, τ_1, τ_2,
(1) $\lfloor F^j(\tau_1) \rfloor_j = \lfloor F^j(\tau_2) \rfloor_j$
(2) $\lfloor F^j(\tau) \rfloor_j = \lfloor F^k(\tau) \rfloor_j$

Proof: (1) By induction.

$$\begin{array}{l} \lfloor F^j(\tau_1) \rfloor_0 = \bot = \lfloor F^j(\tau_2) \rfloor_0. \\ \lfloor F^{j+1}(\tau_1) \rfloor_{j+1} = \\ \lfloor F(F^j(\tau_1)) \rfloor_{j+1} = \\ \lfloor F(\lfloor F^j(\tau_1) \rfloor_j) \rfloor_{j+1} = \\ \lfloor F(\lfloor F^j(\tau_2) \rfloor_j) \rfloor_{j+1} = \\ \lfloor F(F^j(\tau_2)) \rfloor_{j+1} = \\ \lfloor F^{j+1}(\tau_2) \rfloor_{j+1}. \end{array}$$

(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.

Lemma 28 (Type μF)
If F *is well founded, then* μF *is a type.*

Proof: We have to prove that if $\langle k, \Psi, M, v \rangle \in \mu F$ then $M :_k \Psi$. By the definition of μF we have $\langle k, \Psi, M, v \rangle \in F^{k+1}(\bot)$. Since $\bot$ is a type and F is a function from types to types, we have that $F^{k+1}(\bot)$ is a type. Then $\langle k, \Psi, M, v \rangle \in F^{k+1}(\bot)$ allows us to conclude that $M :_k \Psi$.

Next, we must show that μF is closed under memory extension. Suppose that $\langle k, \Psi, M, v \rangle \in \mu F$ and that $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$.

$\langle k, \Psi, M, v \rangle \in \mu F$	
$\langle k, \Psi, M, v \rangle \in F^{k+1}(\bot)$	by def'n of μF
$\langle j, \Psi', M', v \rangle \in F^{k+1}(\bot)$	by def'n 4 (Type)
$\langle j, \Psi', M', v \rangle \in \lfloor F^{k+1}(\bot) \rfloor_{j+1}$	by def'n 1 (Approx)
$\langle j, \Psi', M', v \rangle \in \lfloor F^{j+1}(\bot) \rfloor_{j+1}$	by Lemma 27
$\langle j, \Psi', M', v \rangle \in F^{j+1}(\bot)$	by def'n 1 (Approx)
$\langle j, \Psi', M', v \rangle \in \mu F$	by def'n of μF

□

Lemma 11 (Type $\mathbf{ref}$)
If τ *is a type then* $\mathbf{ref}\ \tau$ *is a type.*

Proof: Given in Section 3. □

Lemma 12 (Type $\forall F$)
If F *is a function from types to types then* $\forall F$ *is a type.*

Proof: Given in Section 3. □

Lemma 29
If $\mathsf{type}(\lfloor \tau \rfloor_k)$ *and* $j \leq k$ *then* $\mathsf{type}(\lfloor \tau \rfloor_j)$.

Proof: Immediate from definitions 4 (Type) and 1 (Approx). □

Lemma 30
If $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$, $i < k$, *and* $i < j$, *then* $(i, \lfloor \Psi \rfloor_i, M) \sqsubseteq (i, \lfloor \Psi' \rfloor_i, M')$.

Proof: Immediate from definitions 3 (Memory Extension) and 1 (Approx). □

Lemma 31 (Type $\exists F$)
If F is a nonexpansive functional then $\exists F$ is a type.

Proof: We must show that if $\langle k, \Psi, M, v\rangle \in \exists F$ then $M :_k \Psi$, but this is follows directly from the definition of $\exists F$.

To prove that $\exists F$ is closed under memory extension, suppose that $\langle k, \Psi, M, v\rangle \in \exists F$ and $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. Note that $\langle k, \Psi, M, v\rangle \in \exists F$ implies that v is of the form $\mathtt{pack}\, v'$ where v' is a value. We must show that $\langle j, \Psi', M', \mathtt{pack}\, v'\rangle \in \exists F$. First, we have to show $M' :_j \Psi'$. But this is immediate from $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$. Second, we must prove the existence of some set τ such that $\mathsf{type}(\lfloor\tau\rfloor_j)$. From $\langle k, \Psi, M, \mathtt{pack}\, v'\rangle \in \exists F$ it follows by the definition of $\exists F$ that there exists some τ such that $\mathsf{type}(\lfloor\tau\rfloor_k)$. Since $j \leq k$ it follows by Lemma 29 that $\mathsf{type}(\lfloor\tau\rfloor_j)$.

Finally, let $i < j$; we must show $\langle i, \lfloor\Psi'\rfloor_i, M', v'\rangle \in F(\tau)$. Since $i < j$ and $j \leq k$ we have $i < k$, and from $\langle k, \Psi, M, \mathtt{pack}\, v'\rangle \in \exists F$ we have $\langle i, \lfloor\Psi\rfloor_i, M, v'\rangle \in F(\tau)$. Since F is a function from types to types and $\lfloor\tau\rfloor_j$ is a type, $F(\lfloor\tau\rfloor_j)$ is a type. It follows that $\lfloor F(\lfloor\tau\rfloor_j)\rfloor_j$ is a type (by definition 1 (Approx)). Since F is nonexpansive, we have that $\lfloor F(\lfloor\tau\rfloor_j)\rfloor_j = \lfloor F(\tau)\rfloor_j$. Hence $\lfloor F(\tau)\rfloor_j$ is a type. Now, since $\langle i, \lfloor\Psi\rfloor_i, M, v'\rangle \in F(\tau)$ and $i < j$, it follows that $\langle i, \lfloor\Psi\rfloor_i, M, v'\rangle \in \lfloor F(\tau)\rfloor_j$. Furthermore, since $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$, $i < j$, and $i < k$, by Lemma 30 we may conclude that $(i, \lfloor\Psi\rfloor_i, M) \sqsubseteq (i, \lfloor\Psi'\rfloor_i, M')$. Since types are closed under memory extension, from $\langle i, \lfloor\Psi\rfloor_i, M, v'\rangle \in \lfloor F(\tau)\rfloor_j$ and $\mathsf{type}(\lfloor F(\tau)\rfloor_j)$ we may conclude that $\langle i, \lfloor\Psi'\rfloor_i, M', v'\rangle \in \lfloor F(\tau)\rfloor_j$. But since $i < j$ it follows that $\langle i, \lfloor\Psi'\rfloor_i, M', v'\rangle \in F(\tau)$. □

B Proofs of Typing Rules

We prove each of the type-checking rules given in Figure 2 as theorems.

Lemma 32 (Closed Application)
If e_1 and e_2 are closed terms and τ_1 and τ_2 are type sets such that $e_1 :_{k,\Psi,M} \tau_1 \to \tau_2$ and $e_2 :_{k,\Psi,M} \tau_1$ then $(e_1\, e_2) :_{k,\Psi,M} \tau_2$.

Proof: Since $e_1 :_{k,\Psi,M} \tau_1 \to \tau_2$ and $e_2 :_{k,\Psi,M} \tau_1$ we immediately have that both (M, e_1) and (M, e_2) are safe for k steps and that if (M, e_1) reduces to some (M', v_1), where v_1 is a value, in fewer than k steps, the value v_1 must be a lambda expression. Hence, the state $(M, (e_1\, e_2))$ either reduces for k steps without any top-level beta-reduction, or there exists a lambda expression $\lambda x.e$, a value v, and a memory M' such that $(M, (e_1\, e_2)) \mapsto^j (M', (\lambda x.e)\, v)$ with $j < k$. In the first case we have that $(M, (e_1\, e_2))$ is safe for k steps and does not generate a value in less than k steps and hence $(e_1\, e_2) :_{k,\Psi,M} \tau_2$ (for any τ_2). In the second case, since $e_1 :_{k,\Psi,M} \tau_1 \to \tau_2$ and $e_2 :_{k,\Psi,M} \tau_1$, definition 5 (Expr : Type) implies that there exists a memory typing Ψ' such that $(k, \Psi, M) \sqsubseteq (k - j, \Psi', M')$, $\langle k - j, \Psi', M', \lambda x.e\rangle \in \tau_1 \to \tau_2$, and $\langle k - j, \Psi', M', v\rangle \in \tau_1$. Pick memory typing $\lfloor\Psi'\rfloor_{k-j-1}$. Then the following memory extension (where we simply forget some information) holds: $(k - j, \Psi', M') \sqsubseteq (k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M')$. Since $\langle k - j, \Psi', M', v\rangle \in \tau_1$ and τ_1 is a type — i.e., by definition 4 (Type) τ_1 is closed under memory extension — we may now conclude that $\langle k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in \tau_1$. The definition of $\to$ then implies that $e[v/x] :_{k-j-1,\lfloor\Psi'\rfloor_{k-j-1},M'} \tau_2$. But we now have $(M, (e_1\, e_2)) \mapsto^{j+1} (M', e[v/x])$ and $(k, \Psi, M) \sqsubseteq (k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M')$, as well as $e[v/x] :_{k-j-1,\lfloor\Psi'\rfloor_{k-j-1},M'} \tau_2$. These three statements imply $(e_1\, e_2) :_{k,\Psi,M} \tau_2$.

□

Theorem 33 (Application)
If Γ is a context, e_1 and e_2 are (possibly open) terms, and τ_1 and τ_2 are types such that $\Gamma \vDash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash e_2 : \tau_1$ then $\Gamma \vDash (e_1 e_2) : \tau_2$

Proof: We must prove that under the premises of the theorem and for any $k \geq 0$ we have $\Gamma \vDash_k (e_1\, e_2) : \tau_2$. More specifically, for any M, Ψ, and σ such that $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$ we must show $\sigma(e_1\, e_2) :_{k,\Psi,M} \tau_2$. By the premises of the theorem we have $\sigma(e_1) :_{k,\Psi,M} \tau_1 \to \tau_2$ and $\sigma(e_2) :_{k,\Psi,M} \tau_1$. The result now follows from Lemma 32.
□

Theorem 34 (Abstraction)
Let Γ be a context, let τ_1 and τ_2 be types, and let $\Gamma[x := \tau_1]$ be the context that is identical to Γ except that it maps x to τ_1. If $\Gamma[x := \tau_1] \vDash e : \tau_2$ then $\Gamma \vDash \lambda x.e : \tau_1 \to \tau_2$.

Proof: As in theorem 33 we must show under the premises of the theorem that for any $k \geq 0$ and M, Ψ and σ such that $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$ we have $\sigma(\lambda x.e) :_{k,\Psi,M} \tau_1 \to \tau_2$. Suppose $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$. Let $j < k$ and v, Ψ', and M' be such that $(k, \Psi, M) \sqsubseteq (j, \Psi', M')$ and $\langle j, \Psi', M', v\rangle \in \tau_1$. By definition of $\to$ it now suffices to show that $\sigma(e[v/x]) :_{j,\Psi',M'} \tau_2$. Let $\sigma\,[x := v]$ be the substitution identical to σ except that it maps x to v. Since the codomain of Γ contains types (which are closed under memory extension), and since $v :_{j,\Psi',M'} \tau_1$, we now have that $\sigma\,[x := v] :_{j,\Psi',M'} \Gamma\,[x := \tau_1]$. By the last premise of the theorem it follows that $\sigma\,[x := v](e) :_{j,\Psi',M'} \tau_2$. But this implies $\sigma(e[v/x]) :_{j,\Psi',M'} \tau_2$. □

Definition 35 (Extend Memory Typing)

$$\mathsf{extmemtype}(k, \Psi, \ell, \tau) \;=\; (\ell \mapsto \lfloor\tau\rfloor_k) \;\cup\; \lfloor\Psi\rfloor_k$$

Lemma 36
If τ is a type such that $\langle k, \Psi, M, v\rangle \in \tau$ where $M :_k \Psi$, $\ell \notin dom(\Psi)$ and $\ell \notin dom(M)$, then

1. $(k, \Psi, M) \sqsubseteq (k, \mathsf{extmemtype}(k, \Psi, \ell, \tau), M\,[\ell := v])$
2. $M\,[\ell := v] :_k \mathsf{extmemtype}(k, \Psi, \ell, \tau)$

Proof: We prove (1) and (2) simultaneously by induction on k.
For (1) we must show the following:
(a) $k \leq k$. Immediate.
(b) $\forall \ell' \in dom(\Psi)$.
$$\lfloor\Psi\rfloor_k(\ell') = \lfloor\mathsf{extmemtype}(k, \Psi, \ell, \tau)\rfloor_k(\ell').$$
Trivial by observing that since $\ell' \neq \ell$, by the definiton of extmemtype, $\lfloor\mathsf{extmemtype}(k, \Psi, \ell, \tau)\rfloor_k(\ell') = \lfloor\Psi\rfloor_k(\ell')$.
(c) $M\,[\ell := v] :_k \mathsf{extmemtype}(k, \Psi, \ell, \tau)$. From (2).

For (2) we must show the following:
(a) $dom(\mathsf{extmemtype}(k, \Psi, \ell, \tau)) \subseteq dom(M\,[\ell := v])$.
Follows easily from premises.
(b) $\mathsf{finite}(dom(M\,[\ell := v]))$.
Follows easily from premises.
(c) $\forall j < k.\ \forall \ell' \in dom(\mathsf{extmemtype}(k, \Psi, \ell, \tau))$.
$$\langle j, \lfloor\mathsf{extmemtype}(k, \Psi, \ell, \tau)\rfloor_j, M\,[\ell := v], M\,[\ell := v](\ell')\rangle \in \lfloor\mathsf{extmemtype}(k, \Psi, \ell, \tau)\rfloor_k(\ell').$$

Case $\ell' \neq \ell$:

- $\langle j, \lfloor\Psi\rfloor_j, M, M(\ell')\rangle \in \lfloor\Psi\rfloor_k(\ell')$ by $M :_k \Psi$.
- By IH (1), since $j < k$, $(j, \lfloor\Psi\rfloor_j, M) \sqsubseteq (j, \lfloor\mathsf{extmemtype}(j, \lfloor\Psi\rfloor_j, \ell, \tau)\rfloor_j, M\,[\ell := v])$.
- Since $\mathsf{type}(\Psi(\ell'))$ $\langle j, \lfloor\mathsf{extmemtype}(j, \lfloor\Psi\rfloor_j, \ell, \tau)\rfloor_j, M\,[\ell := v], M(\ell')\rangle \in \lfloor\Psi\rfloor_k(\ell')$. But by definition of extmemtype we have $\lfloor\Psi\rfloor_k(\ell') = \lfloor\mathsf{extmemtype}(k, \Psi, \ell, \tau)\rfloor_k(\ell')$.

Case $\ell' = \ell$:

- $M\,[\ell := v](\ell') = v$.
- Since $\langle k, \Psi, M, v\rangle \in \tau$ and $j < k$, we have $\langle j, \lfloor\Psi\rfloor_j, M, v\rangle \in \lfloor\tau\rfloor_k$.
- The rest of the proof is similar to case $\ell' \neq \ell$; we note that $\lfloor\mathsf{extmemtype}(k, \Psi, \ell, \tau)\rfloor_k(\ell') = \lfloor\tau\rfloor_k$.

□

Lemma 37 (Closed New)
If e is a closed term and τ is a type set such that $e :_{k,\Psi,M} \tau$ then $\mathtt{new}(e) :_{k,\Psi,M} \mathbf{ref}\ \tau$.

Proof: Since τ is a type by Lemma 11 we have that $\mathbf{ref}\ \tau$ is a type. Then it follows from $e :_{k,\Psi,M} \tau$ that (M, e) is safe for k steps. Hence, if the state $(M, \mathtt{new}(e))$ reduces for k steps without reaching a state of the form $(M', \mathtt{new}(v))$ then it follows that $\mathtt{new}(e) :_{k,\Psi,M} \mathbf{ref}\ \tau$. So we assume, without loss of generality, that $(M, \mathtt{new}(e)) \mapsto^j (M', \mathtt{new}(v))$ for some $j < k$, memory M', and value v. Since $e :_{k,\Psi,M} \tau$, definition 5 (Expr : Type) implies that there exists a Ψ' such that we have (1) and (2) below.

1. $(k, \Psi, M) \sqsubseteq (k - j, \Psi', M')$
2. $\langle k - j, \Psi', M', v\rangle \in \tau$
3. From (1) and definitions 3 and 2 (Memory Extension and Well-typed Memory) we have that M' is finite. It follows that there exists some location $\ell \notin dom(M')$. Hence, we may assume that $(M', \mathtt{new}(v)) \mapsto (M'\,[\ell := v], \ell)$.
4. From (2) it follows that $M' :_{k-j} \Psi'$ (since $\mathsf{type}(\tau)$).
5. Pick $\Psi'' = \mathsf{extmemtype}(k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, \ell, \tau)$. Note that $\lfloor\Psi''\rfloor_{k-j-1} = \Psi''$ by the definition of extmemtype.
6. The following information-forgetting memory extension holds: $(k - j, \Psi', M') \sqsubseteq (k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M')$.
7. From (2) and (6) and the premise that τ is a type, we have $\langle k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in \tau$. From (4) it follows that $M' :_{k-j-1} \lfloor\Psi'\rfloor_{k-j-1}$ and from (3) we have $\ell \notin dom(M')$ and $\ell \notin dom(\lfloor\Psi'\rfloor_{k-j-1})$. Finally, from (5) we have $\Psi'' = \mathsf{extmemtype}(k - j - 1, \Psi', \ell, \tau)$. Hence, we may use Lemma 36 to conclude the following:
   (a) $(k - j - 1, \lfloor\Psi'\rfloor_{k-j-1}, M') \sqsubseteq (k - j - 1, \Psi'', M'\,[\ell := v])$.
   (b) $M'\,[\ell := v] :_{k-j-1} \Psi''$.
8. We must show that $\lfloor\Psi''\rfloor_{k-j-1}(\ell) = \lfloor\tau\rfloor_{k-j-1}$. But $\Psi'' = \mathsf{extmemtype}(k - j - 1, \Psi', \ell, \tau)$ and the definition of extmemtype implies $\lfloor\Psi''\rfloor_{k-j-1} = \Psi''$. Hence, we must show $\Psi''(\ell) = \lfloor\tau\rfloor_{k-j-1}$ which is immediate from the definition of extmemtype.
9. Let $i < (k - j - 1)$. We must show that $\langle i, \lfloor\Psi''\rfloor_i, M'\,[\ell := v], M'\,[\ell := v](\ell)\rangle \in \tau$. By definition 35 (extmemtype), $\ell \in dom(\Psi'')$. Then, from (7b) we can conclude that $\langle i, \lfloor\Psi''\rfloor_i, M'\,[\ell := v], M'\,[\ell := v](\ell)\rangle \in \lfloor\Psi''\rfloor_{k-j-1}(\ell)$. Then, since $\lfloor\Psi''\rfloor_{k-j-1}(\ell) = \lfloor\tau\rfloor_{k-j-1}$ (as we proved in step (8)), and since $i < (k - j - 1)$, it follows that $\langle i, \lfloor\Psi''\rfloor_i, M'\,[\ell := v], M'\,[\ell := v](\ell)\rangle \in \tau$.
10. From (7), (8), and (9) it follows that $\langle k - j - 1, \Psi'', M'\,[\ell := v], \ell\rangle \in \mathbf{ref}\ \tau$.

Finally, we have the following:

- $(M, \mathtt{new}(e)) \mapsto^{j+1} (M'\,[\ell := v], \ell)$ (from $(M, \mathtt{new}(e)) \mapsto^j (M', \mathtt{new}(v))$ and (3));

- $(k, \Psi, M) \sqsubseteq (k-j-1, \Psi'', M'[\ell := v])$ (from (1), (6), (7a), and transitivity of $\sqsubseteq$);
- $\langle k-j-1, \Psi'', M'[\ell := v], \ell \rangle \in \mathbf{ref}\ \tau$ (from (10)).

These three statements imply that $\mathtt{new}(e) :_{k,\Psi,M} \mathbf{ref}\ \tau$. □

The type inference theorem for expressions of the form $\mathtt{new}(e)$ now follows from Lemma 37 in the same manner that Theorem 33 follows from Lemma 32.

Lemma 38 (Closed Dereferencing)
If e is a closed term and τ is a type set such that $e :_{k,\Psi,M} \mathbf{ref}\ \tau$ then $!\,e :_{k,\Psi,M} \tau$.

Proof: Since τ is a type by Lemma 11 we have that $\mathbf{ref}\ \tau$ is a type. Then it follows from $e :_{k,\Psi,M} \mathbf{ref}\ \tau$ that (M, e) is safe for k steps. Hence, the state $(M, !\,e)$ either reduces for k steps without reaching a state of the form $(M', !\,v)$, in which case $!\,e :_{k,\Psi,M} \tau$ (for any τ), or $(M, !\,e) \mapsto^j (M', !\,\ell)$ where $\ell \in dom(M')$ and $j < k$. In the latter case, since $e :_{k,\Psi,M} \mathbf{ref}\ \tau$, definition 5 (Expr : Type) implies that $(k, \Psi, M) \sqsubseteq (k-j, \Psi', M')$ and $\langle k-j, \Psi', M', \ell \rangle \in \mathbf{ref}\ \tau$. By the definition of $\mathbf{ref}$ it follows that $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M', M'(\ell) \rangle \in \tau$. Also, since τ is a type, $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M', M'(\ell) \rangle \in \tau$ implies that $M' :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$. Then the following memory extension (where we forget some information) holds: $(k-j, \Psi', M') \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M')$. By Lemma 9 ($\sqsubseteq$ transitive) it follows that $(k, \Psi, M) \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M')$. But we also have $(M, !\,e) \mapsto^{j+1} (M', M'(\ell))$ (since $(M, !\,v) \mapsto (M', M'(\ell))$), and $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M', M'(\ell) \rangle \in \tau$. These three statements imply $!\,e :_{k,\Psi,M} \tau$.

□

The type inference theorem for expressions of the form $!\,e$ now follows from Lemma 38 in the same manner that Theorem 33 follows from Lemma 32.

Lemma 39
If τ is a type such that $\langle k, \Psi, M, v \rangle \in \tau$ where $M :_k \Psi$, $\ell \in dom(\Psi)$ and $\ell \in dom(M)$, then

1. $(k, \Psi, M) \sqsubseteq (k, \Psi, M[\ell := v])$
2. $M[\ell := v] :_k \Psi$

Proof: Simultaneously by induction on k. The proof is similar to that of Lemma 36. □

Lemma 40 (Closed Assignment)
If e_1 and e_2 are closed terms and τ is a type set such that $e_1 :_{k,\Psi,M} \mathbf{ref}\ \tau$ and $e_2 :_{k,\Psi,M} \tau$ then $e_1 := e_2 :_{k,\Psi,M} \mathbf{unit}$.

Proof: Since τ is a type, by Lemma 11 $\mathbf{ref}\ \tau$ is a type. Then, it follows from $e_1 :_{k,\Psi,M} \mathbf{ref}\ \tau$ and $e_2 :_{k,\Psi,M} \tau$ that (M, e_1) and (M, e_2) are both safe for k steps and that if (M, e_1) reduces to some (M', v_1) in fewer than k steps, the value v_1 must be a location. Hence, the state $(M, e_1 := e_2)$ either reduces for k steps without reaching a state of the form $(M', v_1 := v_2)$ where v_1 and v_2 are values, or there exists a location ℓ and a memory M' such that $\ell \in dom(M')$, $j < k$, and $(M, e_1 := e_2) \mapsto^j (M', \ell := v) \mapsto (M'[\ell := v], \mathbf{0})$. In the first case we have that $(M, e_1 := e_2)$ is safe for k steps and does not produce a value in less than k steps and hence $e_1 := e_2 :_{k,\Psi,M} \mathbf{unit}$. In the second case, $e_1 :_{k,\Psi,M} \mathbf{ref}\ \tau$ and $e_2 :_{k,\Psi,M} \tau$ imply that $(k, \Psi, M) \sqsubseteq (k-j, \Psi', M')$ and $\langle k-j, \Psi', M', \ell \rangle \in \mathbf{ref}\ \tau$ and $\langle k-j, \Psi', M', v \rangle \in \tau$.

We will now show $(k-j, \Psi', M') \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M'[\ell := v])$.

1. The following information-forgetting memory extension holds: $(k-j, \Psi', M') \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M')$.
2. Since τ is a type, from $\langle k-j, \Psi', M', v \rangle \in \tau$ and (1) it follows that $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M', v \rangle \in \tau$.
3. From (2) and type(τ) we have $M' :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$.
4. From (2), (3), type(τ), $\ell \in dom(M')$, and $\ell \in dom(\lfloor \Psi' \rfloor_{k-j-1})$, by Lemma 39 we may conclude the following:
   (a) $(k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M') \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M'[\ell := v])$.
   (b) $M'[\ell := v] :_{k-j-1} \lfloor \Psi' \rfloor_{k-j-1}$.

Finally we have the following:

- $(k, \Psi, M) \sqsubseteq (k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M'[\ell := v])$ (by Lemma 9);
- $(M, e_1 := e_2) \mapsto^{j+1} (M'[\ell := v], \mathbf{0})$;
- $\langle k-j-1, \lfloor \Psi' \rfloor_{k-j-1}, M'[\ell := v], \mathbf{0} \rangle \in \mathbf{unit}$.

These three statements imply $e_1 := e_2 :_{k,\Psi,M} \mathbf{unit}$. □

The type inference theorem for expressions of the form $e_1 := e_2$ now follows from Lemma 40 in the same manner that Theorem 33 follows from Lemma 32.

Theorem 16 (Type Abstraction)
Let Γ be a context and let F be a nonexpansive type functional. If $\Gamma \models e : F(\tau)$ for any type set τ, then $\Gamma \models \Lambda.e : \forall F$.

Proof: Given in Section 4. □

Theorem 14 (Type Application)
Let Γ be a context and let F be a function from types to types. If $\Gamma \models e : \forall F$ then, forall τ such that τ is a type set, $\Gamma \models e[\,] : F(\tau)$.

Proof: Given in Section 4. □

Lemma 41 (Closed Pack)
If e is a closed term, τ is a type set, and F is a function from types to types such that $e :_{k,\Psi,M} F(\tau)$ then $\mathtt{pack}\, e :_{k,\Psi,M} \exists F$.

Proof: Since τ is a type and F is a function from types to types it follows that $F(\tau)$ is a type. Then since $e :_{k,\Psi,M} F(\tau)$ we have that (M, e) is safe for k steps. If the state $(M, \mathtt{pack}\, e)$ reduces for k steps without reaching a state of the form $(M', \mathtt{pack}\, v)$, then we may conclude that $\mathtt{pack}\, e :_{k,\Psi,M} \exists F$. Hence, we may assume without loss of generality that $(M, \mathtt{pack}\, e) \mapsto^j (M', \mathtt{pack}\, v)$ where $j < k$ and v is a value. Since $e :_{k,\Psi,M} F(\tau)$, by definition 5 (Expr : Type) there exists a memory typing Ψ' such that $(k, \Psi, M) \sqsubseteq (k-j, \Psi', M')$ and $\langle k-j, \Psi', M', v\rangle \in F(\tau)$.

We now show that $\langle k-j, \Psi', M', \mathtt{pack}\, v\rangle \in \exists F$. The proof is in three parts. First, from $(k, \Psi, M) \sqsubseteq (k-j, \Psi', M')$ we immediately have that $M' :_{k-j} \Psi'$. Second, from type(τ) (which is a premise of the lemma) there exists a set τ such that type$(\lfloor\tau\rfloor_{k-j})$. Third, suppose $i < (k-j)$. We must show that $\langle i, \lfloor\Psi'\rfloor_i, M', v\rangle \in F(\tau)$. Note that the following information-forgetting memory extension holds: $(k-j, \Psi', M') \sqsubseteq (i, \lfloor\Psi'\rfloor_i, M')$. Since we already have that $\langle k-j, \Psi', M', v\rangle \in F(\tau)$ and that $F(\tau)$ is a type, we can conclude that $\langle i, \Psi', M', v\rangle \in F(\tau)$. Hence we have shown $\langle k-j, \Psi', M', \mathtt{pack}\, v\rangle \in \exists F$.

Finally, we have $(M, \mathtt{pack}\, e) \mapsto^j (M', \mathtt{pack}\, v)$, $(k, \Psi, M) \sqsubseteq (k-j, \Psi', M')$, and $\langle k-j, \Psi', M', \mathtt{pack}\, v\rangle \in \exists F$. These three statements imply that $\mathtt{pack}\, e :_{k,\Psi,M} \exists F$. □

The type inference theorem for pack now follows from Lemma 41 in the same manner that Theorem 33 follows from Lemma 32.

Theorem 42 (Open)
Let Γ be a context and let F be a nonexpansive type functional. If e_1 and e_2 are (possibly open) terms such that $\Gamma \vDash e_1 : \exists F$ and $\Gamma\,[x := F(\tau)] \vDash e_2 : \tau_2$ for any type set τ, then $\Gamma \vDash \mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2 : \tau_2$.

Proof: We must prove under the premises of the theorem for any $k \geq 0$ and M, Ψ, and σ such that $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$, that we have $\sigma(\mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2) :_{k,\Psi,M} \tau_2$.

Suppose $M :_k \Psi$ and $\sigma :_{k,\Psi,M} \Gamma$. By the premises of the theorem we have $\sigma(e_1) :_{k,\Psi,M} \exists F$. Now, since F is a nonexpansive functional by Lemma 31 we have that $\exists F$ is a type. It follows that the state $(M, \sigma(e_1))$ is safe for k steps and that if $(M, \sigma(e_1))$ generates a value in fewer than k steps, that value must be of the form $\mathtt{pack}\, v$ where v is a value. Hence, if the state $(M, \sigma(\mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2))$ does not reduce to a state of the form $(M, \sigma(\mathtt{open}\, (\mathtt{pack}\, v)\, \mathtt{as}\, x\, \mathtt{in}\, e_2))$ in fewer than k steps, then we immediately have $\sigma(\mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2) :_{k,\Psi,M} \tau_2$. So without loss of generality, we can assume that $(M, \sigma(\mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2)) \mapsto^j (M', \sigma(\mathtt{open}\, (\mathtt{pack}\, v)\, \mathtt{as}\, x\, \mathtt{in}\, e_2))$ for $j < k$ and memory M'. Then the operational semantics in Figure 1 allows us to conclude that $(M, \sigma(e_1)) \mapsto^j (M', \sigma(\mathtt{pack}\, v))$ (where $\sigma(\mathtt{pack}\, v) = \mathtt{pack}\, v$ since a value v has no free variables). Since $\sigma(e_1) :_{k,\Psi,M} \exists F$, by definition 5 (Expr : Type) there exists a memory typing Ψ' such that $(k, \Psi, M) \sqsubseteq (k-j, \Psi', M')$ and $\langle k-j, \Psi', M', \mathtt{pack}\, v\rangle \in \exists F$. Then by the definition of $\exists F$ we have that there exists a set τ such that $\lfloor\tau\rfloor_{k-j}$ is a type and $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in F(\tau)$.

By the premises of the theorem, since $\lfloor\tau\rfloor_{k-j}$ is a type, we have $\Gamma\,[x := F(\lfloor\tau\rfloor_{k-j})] \vDash e_2 : \tau_2$. To make further use of this statement, we will first have to show that $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in F(\lfloor\tau\rfloor_{k-j})$. From $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in F(\tau)$ and definition 1 (Approx) we can conclude that $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in \lfloor F(\tau)\rfloor_{k-j}$ (since $(k-j-1) < (k-j)$). By the premises of the theorem F is nonexpansive, which implies that $\lfloor F(\tau)\rfloor_{k-j} = \lfloor F(\lfloor\tau\rfloor_{k-j})\rfloor_{k-j}$. So we have $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in \lfloor F(\lfloor\tau\rfloor_{k-j})\rfloor_{k-j}$. But this implies (since $(k-j-1) < (k-j)$) that $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in F(\lfloor\tau\rfloor_{k-j})$ as we wanted to show.

Let $\sigma\,[x := v]$ be the substitution that is identical to σ except that it maps x to v. Recall that the codomain of Γ contains types (which are closed under memory extension). Note that the following information-forgetting memory extension holds: $(k-j, \Psi', M') \sqsubseteq (k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M')$. Then by Lemma 9 ($\sqsubseteq$ transitive) we have $(k, \Psi, M) \sqsubseteq (k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M')$. This, together with $\langle k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M', v\rangle \in F(\lfloor\tau\rfloor_{k-j})$ (which we proved above), allows us to conclude that $\sigma\,[x := v] :_{k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M'} \Gamma\,[x := F(\lfloor\tau\rfloor_{k-j})]$. Hence from $\Gamma\,[x := F(\lfloor\tau\rfloor_{k-j})] \vDash e_2 : \tau_2$ it follows that $\sigma\,[x := v](e_2) :_{k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M'} \tau_2$. But this implies $\sigma(e_2[v/x]) :_{k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M'} \tau_2$.

Finally, we have $(M, \sigma(\mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2)) \mapsto^{j+1} (M', \sigma(e_2[v/x]))$; $(k, \Psi, M) \sqsubseteq (k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M')$; and $\sigma(e_2[v/x]) :_{k-j-1, \lfloor\Psi'\rfloor_{k-j-1}, M'} \tau_2$. Hence, we can conclude that $\sigma(\mathtt{open}\, e_1\, \mathtt{as}\, x\, \mathtt{in}\, e_2) :_{k,\Psi,M} \tau_2$. □