A Stratified Semantics of General References Embeddable in Higher-Order Logic[*]

(Extended Abstract)

Amal J. Ahmed[†] Andrew W. Appel[‡] Roberto Virga[§]

Princeton University

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

Abstract

We demonstrate a semantic model of general references — that is, mutable memory cells that may contain values of any (statically-checked) closed type, including other references. Our model is in terms of execution sequences on a von Neumann machine; thus, it can be used in a Proof-Carrying Code system where the skeptical consumer checks even the proofs of the typing rules. The model allows us to prove a frame-axiom introduction rule that allows locality of specification and reasoning, even in the event of updates to aliased locations. Our proof is machine-checked in the Twelf metalogic.

1 Introduction

Proof-carrying code is a framework for proving the safety of machine-language programs with a machine-checkable proof. In conventional PCC systems [19, 18], proofs are written in a logic with a built-in understanding of a particular type system; that is, each inference rule of the type system is an axiom of the logic. In foundational PCC, introduced by Appel and Felty [4], the only axiom besides the axioms of higher-order logic and arithmetic is the definition of the state-transition relation of the target architecture. The semantics of everything else (safety, types, etc.) must be modeled in terms of possible state transitions. For very simple type systems, with immutable references, no data structure creation, and no recursive types, such models are easy to construct. Appel and Felty [4] have shown how to extend this to allocation of immutable values, covariant recursive types, function pointers, and quantified types. Appel and McAllester [5] further extend this to contravariant recursive types. Our new result is an extension of all the previous type systems to mutable references, where reference cells can contain values of any type, including functions and other references.

Almost [12] all practical programming languages use mutable references; object-oriented languages (such as Java) and functional languages (such as ML) permit references to contain values of arbitrary (statically-checked) type. Therefore, general references are essential in our plans to build PCC systems for practical languages. Our model can handle the full language of ML or Java references, including cyclic data structures, covariant and contravariant recursive data types, and gives a detailed semantics for the initialization, allocation and update of data structures in memory. The foundational PCC consumer need not know, or trust, the typing rules in advance. This means that we must provide a machine-checkable proof of these rules; for this we use a semantic model.

The denotational semantics of general references have posed a challenge to semanticists for years [8]. Recently, however, some solutions have emerged [1, 15]. An important aspect of our model, which is a mix of denotational and operational semantics, is that it is immediately useful as a formalism for proving properties of machine-language programs. The formalism allows locality of specification and reasoning, even in the event of memory updates, and even in the presence of aliasing.

In a typical syntactic theory of references we have judgments of the form $\Psi, \Gamma \vdash x : \tau$ (where Ψ is a mapping from locations to types). In the Appel-Felty semantics, a type is a predicate on a set of allocated locations a, a memory m, and a root-pointer x, where a is simply a set of addresses. It seems natural to generalize a to serve the role of Ψ, thus extending Appel-Felty to model general references. Unfortunately, this leads to a circularity. The main contribution of this paper is to eliminate this circularity. Our approach con-

[*]To appear in LICS 2002

[†]Supported in part by DARPA Grant F30602-99-1-0519 and by NSF Grant 9972930.

[‡]Supported in part by DARPA Grant F30602-99-1-0519.

[§]Supported by DARPA Grant F30602-99-1-0519.

sists of a stratification of the type universe, together with an interesting use of Gödel numbering as a way to encode the resulting hierarchy of types in higher-order logic. Also, our semantics is based on a possible-worlds model which seems crucial in modeling the intensionality inherent in ML-style references.

2 Foundational proofs of safety

We begin by summarizing the foundational PCC approach to proving the safety of machine-language programs.

Specifying safety. The first step is to build a model of a von Neumann machine, such as the Sparc or the Pentium, and a safety policy. In this model, a machine state comprises a *register bank* and a *memory*, each of which is a function from integers (addresses) to integers (contents).

The execution of an instruction is modeled as a single step of the machine. First, we define each instruction i as a predicate on four arguments (r, m, r', m') such that, given a machine at state (r, m), after execution of instruction i the machine will be at state (r', m'), provided that the execution does not violate the safety policy. For example, if the safety policy requires that "only *writable* addresses may be updated," (where the predicate `writable` is suitably specified as part of the safety policy), we can define the instruction $\mathbf{m}[\mathbf{r_j} + \mathbf{c}] \leftarrow \mathbf{r_i}$ as:

$$\begin{array}{l}\texttt{store}(i, j, c) \;=\; \\ \quad \lambda r, m, r', m'.\ \texttt{writable}(r(j) + c) \;\wedge\; m'(r(j) + c) = r(i) \\ \qquad\qquad \wedge\; (\forall x \neq (r(j) + c).\, m'(x) = m(x)) \;\wedge\; r' = r\end{array}$$

Next, we specify the step relation $(r, m) \mapsto (r', m')$ which formally describes a single instruction execution. It requires the existence of an instruction i and a register bank r'' such that the integer at location $r(\text{PC})$ (PC is the program counter) in memory m decodes to instruction i, updating the register bank r with an incremented program counter produces r'', and finally instruction i safely maps (r'', m) to (r', m'):

$$\begin{array}{l}(r, m) \mapsto (r', m') \;=\; \\ \quad \exists r'', i.\ \texttt{decode}(m(r(\text{PC})), i) \\ \qquad\qquad \wedge\; r'' = r\,[\text{PC} := r(\text{PC}) + 1] \;\wedge\; i(r'', m, r', m')\end{array}$$

where $\;f\,[d := x] \;=\; \lambda i.\ \text{if}\;\; i = d \;\;\text{then}\;\; x \;\;\text{else}\;\; f(i).$ [1]

We model a state in which the real machine would have a next step that violates the safety policy, as a state with no successor in the step relation. Then, proving that a state is safe (written $\texttt{safe}(r, m)$) amounts to showing that there is no path from (r, m) to a state with no successor. To prove a program safe it suffices to show that a state (r, m) where the program is loaded in memory m and the program counter $r(\text{PC})$ points to the first instruction of the program, is a safe state. We say that a state (r, m) is safe to execute for k steps, written $\texttt{safen}(k, r, m)$, if it cannot get stuck within k instructions. Then, we show $\texttt{safe}(r, m)$ by proving $\forall k.\, \texttt{safen}(k, r, m)$ by induction on k, the number of future execution steps.

[1] For details on how to specify instruction encoding and semantics for real machine architectures, see Michael and Appel [16].

Proving programs safe. A program is a sequence of machine instructions at a specific place in memory. 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. The global invariant Γ maps each location in the program to its local invariant. [2] In foundational PCC (and also in Necula [19]) preconditions are expressed using types, e.g., $r(1) : \tau_1 \;\wedge\; m(102) : \tau_5$. A judgment $x : \tau$ in foundational PCC is interpreted as $x :_{k,m} \tau$, which may be read as "x has type τ with respect to memory m to approximation k" or "the assumption that x has type τ cannot be proved wrong within k steps of execution". Program invariants, then, are parametrized by k, r, and m. Appel and McAllester [5] give a formal interpretation of the judgment $\Gamma \vdash \{P\}C\{Q\}$, (where C is an instruction and P and Q are the pre- and postcondition, respectively), as a statement about safe future execution of a program with respect to an induction hypothesis Γ in a foundational PCC system. Under this interpretation, to show $\Gamma \vdash \{P\}C\{Q\}$ it is sufficient to prove two lemmas: progress and approximate preservation. Progress says that when the program counter points to address l, if C is a valid instruction at address l and the invariant at address l holds, then we can safely execute C:

Lemma 1 (Progress)

$$\frac{\begin{array}{ccc}\texttt{decode}(m(l), [\![C]\!]) & \Gamma(l) = P & \Gamma(l+1) = Q \\ r(\text{PC}) = l & P(k, r, m) & k \geq 1\end{array}}{\exists r', m'.\, (r, m) \mapsto (r', m')}$$

Approximate preservation [3] says that if the invariant at address l holds, then executing the instruction at l leads to a state (r', m') such that the invariant $\Gamma(r'(\text{PC}))$ is satisfied in state (r', m') with approximation $k - 1$ (since one execution step was consumed by the execution of C):

Lemma 2 (Approximate Preservation)

$$\frac{\begin{array}{cccc}\texttt{decode}(m(l), [\![C]\!]) & \Gamma(l) = P & \Gamma(l+1) = Q & \\ r(\text{PC}) = l & P(k, r, m) & k \geq 1 & (r, m) \mapsto (r', m')\end{array}}{\Gamma(r'(\text{PC}))(k - 1, r', m')}$$

3 Semantic models of types

To motivate our upcoming model, we present three Hoare triples that read from, initialize, and update memory, respectively, and show type-inference rules required to prove that each triple holds. In a foundational system these

[2] In practice, it would suffice for Γ to map only the entry points of basic blocks to the appropriate invariants.

[3] We use "approximate" to indicate that unlike regular proofs of preservation, here the induction is on the number of *future* execution steps k.

inference rules cannot be added to the logic as axioms; we discuss the semantic models of types from which they can be derived as lemmas.

3.1 An indexed model

Example 1 (Traversal of heap-allocated data)
Consider the following Hoare triple involving an instruction that reads a value from a data structure in memory. The precondition says that this data structure must be a reference cell containing a value of type τ, while the postcondition requires that the value in the destination register has type τ.

$$\begin{array}{l} \{\lambda k, r, m.\, r(2) :_{k,m} \mathsf{ref}\ \tau\} \\ \mathbf{r_3} \leftarrow \mathbf{m}(\mathbf{r_2}) \\ \{\lambda k, r, m.\, r(3) :_{k,m} \tau\} \end{array}$$

Let us prove that the above triple holds with respect to the global invariant Γ. To simplify the exposition, we will concentrate on proving approximate preservation — in our automated proofs, of course, we also prove progress. The step relation increments the program counter and the above instruction is not a control-flow instruction, so $r'(\textsc{pc}) = l+1$ is easily proved. Since we know from the semantics of the load instruction that $m' = m$, to prove $r_3 :_{k-1,m'} \tau$, we can use an inference rule similar to the **Ref Elimination** rule below. This rule says that if x is a pointer to a value of type τ in memory m with index k, then the contents of memory m at address x are of type τ with index $k-1$ (since one execution step is consumed in dereferencing the pointer).

Ref Elimination
$$\frac{x :_{k,m} \mathsf{ref}\ \tau}{m(x) :_{k-1,m} \tau}$$

But where does this inference rule come from? A *type-specialized* PCC system (such as Necula's [19]) would include a similar rule as an axiom. In foundational PCC, however, we build a semantic model of types that allows us to prove this type inference rule as a lemma. A value is a pair (m, x) of a memory m and an integer x (usually an address that can be thought of as the root-pointer to a data structure in memory). The domain of types has elements that are sequences of k-approximations to the value (m, x), where (k, m, x) is in a type τ iff (m, x) is "good enough" for k steps of execution [5]. Then, a program that executes j instructions where $j \leq k$ also *believes* that x has type τ; that is, $x :_{k,m} \tau \Rightarrow \forall j \leq k.\, x :_{j,m} \tau$. Types can then be defined as predicates on (k, m, x) so that the judgment $x :_{k,m} \tau$ is just syntactic sugar for $\tau(k, m, x)$. We can define integer and reference types as follows:

$$\begin{array}{lcl} \mathsf{int}(k,m,x) & = & \texttt{true} \\ (\mathsf{ref}\ \tau)(k,m,x) & = & \texttt{readable}(x) \wedge \forall j{<}k.\, \tau(j,m,m(x)) \end{array}$$

From the definition of ref above, we can immediately prove the **Ref Elimination** rule as a lemma.

By defining a variety of types in this way (Appel and Felty [4] provide an extensive catalog), and using them as building blocks to describe more complicated datatypes, we can reason about the safety of programs that traverse non-trivial data structures — just as long as these data structures are *statically* allocated.

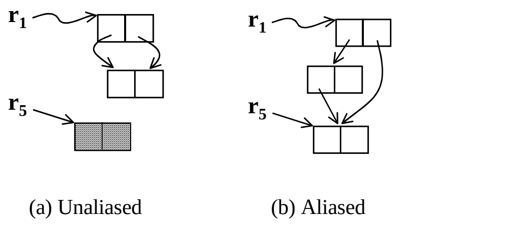


Figure 1. Pointer Aliasing

3.2 A dynamic allocation model

Example 2 (Dynamic heap allocation)
Programs written in a call-by-value pure functional language allocate new data structures on the heap but never update old values. Appel and Felty [4] describe a semantic model that allows us to reason about the safety of such programs. Consider the following Hoare triple for an instruction that creates a new reference cell in memory by writing to a *new*, `writable` memory location, pointed to by register r_5. The situation is depicted in figure 1(a) where all "new" or unallocated memory cells appear shaded.

$$\begin{array}{l} \{\lambda k, r, m.\, r(1) :_{k,m} \tau_1 \wedge r(4) :_{k,m} \tau_2\} \\ \mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4} \\ \{\lambda k, r, m.\, r(1) :_{k,m} \tau_1 \wedge m(r(5)) :_{k,m} \tau_2\} \end{array}$$

Given $r_1 :_{k,m} \tau_1$ and the fact that the store instruction alters memory (i.e., $m' \neq m$), how can we prove $r_1 :_{k,m'} \tau_1$? First, we distinguish between allocated and unallocated locations by maintaining a set a of allocated addresses. We expect that the program's memory allocator module keeps track of which memory addresses it has allocated using some data structure in registers and memory. Thus, from time to time, the set a is computable by some function $a = \texttt{alloc}(r, m)$. At other times, (for example, part way through an allocation), it may be that $a \neq \texttt{alloc}(r, m)$. As a result, a is existentially quantified when it appears in program invariants.

We say that a *state* is a pair (a, m) of an allocset a and a memory m. A value is now a tuple (k, a, m, x) of an index k, a state and a root-pointer and types are, as before, predicates on values. Only `readable` and `writable` locations are added to the allocset. This is accomplished by giving the program's allocator module an initial pool that is a subset of ($\texttt{readable} \cap \texttt{writable}$). The types int and ref are defined as,

$$\begin{array}{lcl} \mathsf{int}(k,a,m,x) & = & \texttt{true} \\ (\mathsf{ref}\ \tau)(k,a,m,x) & = & x \in a \wedge \forall j < k.\, \tau(j,a,m,m(x)) \end{array}$$

Next, we specify that only unallocated locations can be modified, i.e., from a state (a, m) we can get to a state (a, m') if and only if $\forall x. x \in a \Rightarrow m(x) = m'(x)$. In this model, a store instruction never affects existing data structures; hence, we can prove that existing type judgments are preserved across memory updates. This model allows us to prove the following **Initialization Invariance** rule as a lemma. The rule says that when we update an unallocated location, type judgments made with respect to the old memory continue to be valid with respect to the new memory:

Initialization Invariance

$$\frac{x :_{k,a,m} \tau \qquad y \notin a \qquad m' = m\,[y := z]}{x :_{k-1,a,m'} \tau}$$

If we rewrite the invariants of our Hoare triple so that type judgments have the form $x :_{k,a,m} \tau$, using the **Initialization Invariance** rule we can prove the following statement:

$$\begin{array}{l}\{\lambda k, r, m.\, \exists a.\, r(1) :_{a,m} \tau_1 \wedge r(5) \notin a \wedge r(4) :_{a,m} \tau_2\} \\ \mathbf{m(r_5) \leftarrow r_4} \\ \{\lambda k, r, m.\, \exists a.\, r(1) :_{k,a,m} \tau_1 \wedge m(r(5)) :_{k,a,m} \tau_2\}\end{array}$$

3.3 The need for a new model

Example 3 (Mutable data structures)
The model described by Appel and Felty [4] cannot be used to reason about the safety of programs written in an imperative language. That model prohibits updates to allocated memory locations — the store instruction in the following Hoare triple performs such an update:

$$\begin{array}{l}\{\lambda k, r, m.\, \exists a.\, r(1) :_{k,a,m} \tau_1 \\ \qquad\qquad\qquad \wedge\, r(5) :_{k,a,m} \mathsf{ref}\ \tau_2 \wedge r(4) :_{k,a,m} \tau_2\} \\ \mathbf{m(r_5) \leftarrow r_4} \\ \{\lambda k, r, m.\, \exists a.\, r(1) :_{k,a,m} \tau_1 \wedge r(5) :_{k,a,m} \mathsf{ref}\ \tau_2\}\end{array}$$

Consider the scenario illustrated by figure 1(b) — the store instruction updates the location that r_5 points to (thereby modifying the data structure that r_1 points to), so that we cannot know if r_1 has type τ_1 with respect to the modified memory m'. We do not want to rule out situations such as this one where an aliased location is being updated. The **Update Invariance** rule allows us to handle updates even in the presence of aliasing. This rule says that when we update an *allocated* location, type judgments made with respect to the old memory continue to be valid with respect to the new memory. This suggests that writing to an allocated location should be permitted only if the update is type-preserving.

Update Invariance

$$\frac{x :_{k,a,m} \tau \quad y :_{k,a,m} \mathsf{ref}\ \tau' \quad z :_{k,a,m} \tau' \quad m' = m\,[y := z]}{x :_{k-1,a,m'} \tau}$$

Allowing updates of aliased locations while guaranteeing consistency is not an easy task — i.e., proving the **Update Invariance** rule is nontrivial. For foundational PCC, we must devise a new semantic model of types that allows us to prove the **Update Invariance** rule as a lemma. Such a model and a proof of an **Update Invariance** lemma are the main contributions of this paper.

An update to an allocated location must be type-preserving — this means that only values of a certain type may be written at that location. Hence, we require a model that, for each allocated location, keeps track of this type. In the next section we describe why tracking permissible heap updates is tricky.

4 Modeling permissible heap updates

In the semantics of immutable fields described in section 3.2 a type is a predicate on an index k (an integer), a memory m (a function from integers to integers), a set a of allocated addresses (a predicate on integers), and a root-pointer x (an integer). In our object logic, we write the types of these logical objects as,

$$\begin{array}{lcl} \mathit{memory} & = & \mathit{num} \;\rightarrow\; \mathit{num} \\ \mathit{allocset} & = & \mathit{num} \;\rightarrow\; o \\ \mathit{type} & = & \mathit{num} \times \mathit{allocset} \times \mathit{memory} \times \mathit{num} \;\rightarrow\; o \end{array}$$

where o is the type of propositions (`true` or `false`).

4.1 Putting types in the allocset

To allow for the update of existing values we might think of enhancing the allocset a to become a finite map from locations to types: for each allocated address x, we keep track of the type τ of updates allowed at x. As before, a type is a predicate on four arguments (k, a, m, x):

$$\begin{array}{lcl} \mathit{allocset} & = & \mathit{num} \overset{\text{fin}}{\rightarrow} \mathit{type} \\ \mathit{type} & = & \mathit{num} \times \mathit{allocset} \times \mathit{memory} \times \mathit{num} \;\rightarrow\; o \end{array}$$

But there is a problem with this specification: notice that the metalogical type of *type* is recursive, and, furthermore, that it has an inconsistent cardinality: the set of types must be bigger than itself.

4.2 A hierarchy of types

To better understand the problem, let us take a closer look at our *desired* definition of ref. We say $x :_{k,a,m} \mathsf{ref}\ \tau$ if location x is allocated, if the allocset a says that the permissible update type for location x is τ, and if the value in memory at location x is of type τ with index j for $j < k$: [4]

$$(\mathsf{ref}\ \tau)(k, a, m, x) \;=\; (x, \tau) \in a \;\wedge\; \forall j < k.\, \tau(j, a, m, m(x))$$

Notice that τ is a "smaller" type than ref τ and that to determine the members of ref τ we, in fact, only consider those locations in the allocset whose permissible update types are

[4]Recall that we would like the allocset to be a finite map from locations to types. Since a finite map can be modeled as a relation, we write $(x, \tau) \in a$ rather than $a(x) = \tau$.

"smaller" than ref τ. This suggests a well-foundedness ordering: types in our model should be stratified so that a type at level i relies not on the *entire* allocset, but only on that subset of the allocset that maps locations to types at level j, for $j < i$. This leads us to the following *type* hierarchy:

$$\begin{array}{lcl} \mathit{type}_0 & = & \mathit{unit} \\ \mathit{allocset}_i & = & \mathit{num} \overset{\text{fin}}{\rightarrow} \mathit{type}_i \\ \mathit{type}_{i+1} & = & \mathit{num} \times \mathit{allocset}_i \times \mathit{memory} \times \mathit{num} \;\rightarrow\; o \end{array}$$

By stratifying mutable references we have eliminated the circularity. Unfortunately, the above *type* hierarchy does not fit into higher-order logic. We would like to have a single type of *type* in our object logic, not an infinite number of them.

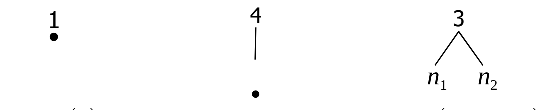
4.3 A hierarchy of Godel numberings of types

To achieve a single type of *type*, we present a solution that replaces the (semantic) type (type_i) in the allocset with its syntax. This syntax is simply a free algebra of type expression terms. To manipulate syntactic types in higher-order logic, we encode them as Gödel numbers (we use this term in a general sense, to mean simply "unique identifiers"). We use a stratified Gödel numbering relation $\mathtt{rep}(i, n, \tau)$ where the Gödel number n represents the type τ at level i. Note that instead of encoding syntactic types using integers, we opted to use Gödel numbers that are finite trees of integers. The types of the relevant logical objects and of the Gödel numbering relation `rep` are as follows:

$$\begin{array}{lcl} \mathit{gnum} & = & \mathit{tree}(\mathit{num}) \\ \mathit{allocset} & = & \mathit{num} \overset{\text{fin}}{\rightarrow} \mathit{gnum} \\ \mathit{pretype} & = & \mathit{num} \times \mathit{allocset} \times \mathit{memory} \times \mathit{num} \;\rightarrow\; o \\ \mathtt{rep} & : & \mathit{num} \times \mathit{gnum} \times \mathit{pretype} \;\rightarrow\; o \\ \mathit{type} & = & \mathit{num} \;\rightarrow\; \mathit{pretype} \end{array}$$

A type is now a predicate on (i, k, a, m, x) where i is the index of the type in the type hierarchy; this corresponds informally to a logical object of type type_i as in section 4.2. A predicate on (k, a, m, x) is now called a pretype. (Henceforth, we will use σ to range over pretypes and τ to range over types.)

Before we describe our pretypes and their encoding, we need some notation for representing trees of integers. A tree constructor $\mathtt{tree}_i(c_0, t_1, \ldots, t_i)$ returns a tree with integer c_0 at the root and i subtrees $t_1, \ldots, t_i$, for example:



$\mathtt{tree}_0(1)$ $\quad$ $\mathtt{tree}_1(4, \mathtt{tree}_0(1))$ $\quad$ $\mathtt{tree}_2(3, n_1, n_2)$

Pretypes. We now define the pretypes $\overline{\mathsf{int}}$, $\overline{\cup}$, and $\overline{\mathsf{ref}}$, (for a type constructor tycon, $\overline{\mathsf{tycon}}$ denotes the corresponding pretype constructor), and describe how a `rep` relation may be defined for these pretypes. Note that since the definition of $\overline{\mathsf{ref}}$ depends on `rep` (which has not been defined yet), $\overline{\mathsf{ref}}$ should take $\mathtt{rep}(i)$ (for some $i \geq 0$) as an argument. We use ρ to denote the corresponding formal parameter, where the type of ρ is as follows: $\rho : \mathit{gnum} \times \mathit{pretype} \;\rightarrow\; o$. (All subsequent uses of the variable ρ will be of this logical type — i.e. ρ is always used to denote $\mathtt{rep}(i)$ for some $i \geq 0$.)

$$\begin{array}{lcl} \overline{\mathsf{int}}(k, a, m, x) & = & \mathtt{true} \\ (\sigma_1 \mathbin{\overline{\cup}} \sigma_2)(k, a, m, x) & = & \sigma_1(k, a, m, x) \;\vee\; \sigma_2(k, a, m, x) \\ (\overline{\mathsf{ref}}(\rho, \sigma))(k, a, m, x) & = & \exists n.\; (x, n) \in a \;\wedge\; \rho(n, \sigma) \\ & & \qquad \wedge\; \forall j < k.\, \sigma(j, a, m, m(x)) \end{array}$$

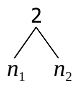
Defining `rep`. We can define, as a formula in higher-order logic, a `rep` predicate that has the following properties:

1. It relates the pretype $\overline{\mathsf{int}}$ to the tree $\mathtt{tree}_0(1)$:

$$\frac{}{\mathtt{rep}(0, \mathtt{tree}_0(1), \overline{\mathsf{int}})}$$

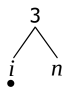
2. It relates $\sigma_1 \mathbin{\overline{\cup}} \sigma_2$ to the tree below as follows:

$$\frac{\mathtt{rep}(i, n_1, \sigma_1) \qquad \mathtt{rep}(i, n_2, \sigma_2)}{\mathtt{rep}(i, \mathtt{tree}_2(2, n_1, n_2), \sigma_1 \mathbin{\overline{\cup}} \sigma_2)}$$



3. It relates the pretype $\overline{\mathsf{ref}}(\rho, \sigma)$ to a tree where one child is a tree with the single root node i. Since ρ encapsulates i, the level of σ in the type hierarchy, i must be part of the Gödel number for $\overline{\mathsf{ref}}(\rho, \sigma)$ (note that $\overline{\mathsf{ref}}(\rho, \sigma)$ is at level $i + 1$ in this hierarchy):

$$\frac{\mathtt{rep}(i, n, \sigma)}{\mathtt{rep}(i + 1, \mathtt{tree}_2(3, \mathtt{tree}_0(i), n), \overline{\mathsf{ref}}(\rho, \sigma))}$$



4. $\mathtt{rep}(i) \subset \mathtt{rep}(i + 1)$:

$$\frac{\mathtt{rep}(i, n, \sigma)}{\mathtt{rep}(i + 1, n, \sigma)}$$

We show the inductive definition in the technical report [3].

Figure 2 illustrates the first few levels of the hierarchy for the pretype constructors $\overline{\mathsf{int}}$, $\overline{\cup}$, and $\overline{\mathsf{ref}}$. Level 0 consists of the Gödel numberings of $\overline{\mathsf{int}}$, $\overline{\mathsf{int}} \mathbin{\overline{\cup}} \overline{\mathsf{int}}$, $(\overline{\mathsf{int}} \mathbin{\overline{\cup}} \overline{\mathsf{int}}) \mathbin{\overline{\cup}} \overline{\mathsf{int}}$, and so on. Let σ^0 denote a pretype that has a Gödel number at level 0. Level 1 consists of Gödel numberings of pretypes σ^0, of pretypes $\overline{\mathsf{ref}}(\mathtt{rep}(0), \sigma^0)$, and of all pretypes in the closure (with respect to $\overline{\cup}$) of the level 1 pretypes.

Figure 2 also describes the structure of the `rep` relation: Here, `base` is a subset of $\mathtt{rep}(0)$ and specifies the Gödel numbers of all primitive types; $\mathtt{closure}(\rho)$ specifies all the types constructible by unioning together types numbered in ρ; $\mathtt{step}(\mathtt{rep}(i))$ defines a subset of $\mathtt{rep}(i{+}1)$; the `closure` of the latter, then, gives us $\mathtt{rep}(i + 1)$.

Types. Having defined `rep`, we can now define the types that correspond to the pretypes defined above:

$$\begin{array}{lcl} \mathsf{int}(i) & = & \overline{\mathsf{int}} \\ (\tau_1 \cup \tau_2)(i) & = & \tau_1(i) \mathbin{\overline{\cup}} \tau_2(i) \\ (\mathsf{ref}\ \tau)(i) & = & \overline{\mathsf{ref}}(\mathtt{rep}(i - 1), \tau(i - 1)) \end{array}$$

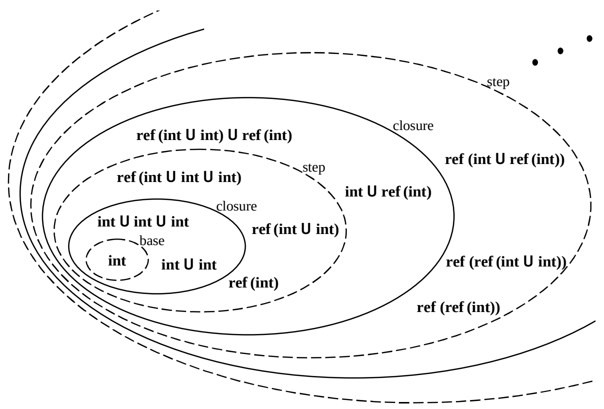


Figure 2. Hierarchy of Godel numberings

By creating a hierarchy of Gödel numbers, we have, in effect, created a hierarchy of types: to determine the elements of a type at level $i + 1$ we need to know the elements of all types at levels 0 through i — for this we use $\texttt{rep}(i)$. By stratifying types we have eliminated the circularity.

5 Possible worlds

The semantics we are presenting is a possible-worlds semantics. Possible-worlds models (or Kripke models [5]) are specified by defining (see Huth and Ryan [13]):

- A set W, whose elements are called *worlds*. In our model, a world corresponds to a state so we have, $W = \mathit{allocset} \times \mathit{memory}$.
- A relation $R \subseteq W \times W$ called the accessibility relation. In our model, this corresponds to the extend-state relation on states (a, m) and (a', m') which describes how we can get from a well-formed (or *valid*) state (a, m) to a valid state (a', m'). Section 5.1 specifies when a state is considered valid and section 5.2 specifies the extend-state relation.
- A labeling function $L : W \to \mathcal{P}(\texttt{Atoms})$ that, given a world w, yields the set of atomic propositions that hold in that world. The atomic propositions p that we are interested in are of the form: "location l may hold a value of pretype σ" where l is an allocated location. Therefore, in our model,
$$L(a, m) = \{(l, \sigma) \mid (l, n) \in a \wedge \exists i.\, \texttt{rep}(i, n, \sigma)\}$$
- The properties that the accessibility relation R should satisfy — these depend on what set of formulas involving p (where p is an atomic proposition) should be valid in the desired model. Without going into further detail, our model requires that the extend-state relation be reflexive and transitive (lemma 6).

[5]Kripke [29] introduced the notion of possible worlds when he developed the model theory for modal propositional logic based on this concept.

5.1 Valid states

The judgment $x :_{k,a,m} \sigma$ says that x has pretype σ for up to k execution steps with respect to the state (a, m). Implicit in this assertion is the assumption that state (a, m) is well-formed or *valid* such that types are guaranteed to be preserved for k steps. A state (a, m) is valid with index k if it satisfies three conditions. First, the allocset a must be a partial function, that is, each location in the allocset should be mapped to only one Gödel number (informally, to only one type). Second, a should only map locations to legitimate Gödel numbers (informally, *valid* types). The predicate $\texttt{godel}$ specifies legitimate Gödel numbers:

$$\texttt{godel}_\rho(n) \;=\; \exists \rho', \sigma.\; \rho \subset \rho' \;\wedge\; \rho'(n, \sigma)$$

The subscript ρ indicates that $\texttt{godel}$ takes $\texttt{rep}(i)$ as an argument. (Note that we could have instead defined $\texttt{godel}_\rho(n)$ as $\exists \sigma. \rho(n, \sigma)$. We explain why the present formulation is useful when we discuss the definition of the type $\texttt{codeptr}$ in section 7.) Third, the type of an allocated cell's contents (with respect to k, a, and m) must match its permissible update type in a; that is, if a maps a location l to pretype σ then $m(l) :_{k,a,m} \sigma$ should hold.) We say that x matches a Gödel number n with respect to a state (a, m) and index k if,

$$\begin{aligned}&\texttt{match}_\rho(k, n, a, m, x) \;=\\ &\quad \forall \rho', \sigma.\; \rho \subset \rho' \;\Rightarrow\; \rho'(n, \sigma) \;\Rightarrow\; \sigma(k, a, m, x)\end{aligned}$$

Definition 3 (Valid State)

$$\begin{aligned}&\texttt{validstate}_\rho(k, a, m) \;=\\ &\quad \forall x, n, n'.\; (x, n) \in a \;\wedge\; (x, n') \in a \;\Rightarrow\; n =_{tree} n'\\ &\quad\; \wedge\; \forall x, n.\; (x, n) \in a \;\Rightarrow\; \texttt{godel}_\rho(n)\\ &\quad\; \wedge\; \forall x, n.\; (x, n) \in a \;\Rightarrow\; \texttt{match}_\rho(k, n, a, m, m(x))\end{aligned}$$

The following property follows from the definition of $\texttt{validstate}$. Informally, the type of the value in an allocated cell matches the type that the allocset says it should have for up to k execution steps:

Lemma 4 (Heap Well-Typed)

$$\frac{(x, n) \in a \qquad \texttt{validstate}_\rho(k, a, m)}{\exists \rho', \sigma.\; \rho \subset \rho' \;\wedge\; \rho'(n, \sigma) \;\wedge\; m(x) :_{k,a,m} \sigma}$$

5.2 Valid state extension

To formally describe the memory and allocset extensions permissible in our model we specify the extend-state relation $(a, m) \sqsubseteq_{\rho,k} (a', m')$ which says that state (a', m') is a valid extension of state (a, m) with index k — or, alternatively, that (a, m) approximates (a', m') for up to k execution steps. State extensions must satisfy three constraints. First, memory cannot be deallocated, so if $x \in \mathrm{dom}(a)$, then we require that $x \in \mathrm{dom}(a')$. (The reason for this restriction is that the extend-state relation must be transitive.) Second, the permissible update type of an allocated location cannot be altered across state extensions, so if $(x, n) \in a$

then $(x, n) \in a'$. Third, the model requires that all memory updates be type-preserving — to enforce this we simply require that state (a', m') be a valid state (which suffices because the last two conditions ensure that all allocated locations are "preserved" under state extension).

Definition 5 (Extend State)
Valid state extension ($\sqsubseteq_{\rho,k}$) is specified as,

$$\begin{array}{l}(a, m) \sqsubseteq_{\rho,k} (a', m') \;=\\ \quad \forall x, n.\; (x, n) \in a \;\Rightarrow\; (x, n) \in a' \\ \quad \wedge\; \mathtt{validstate}_\rho(k, a, m) \;\wedge\; \mathtt{validstate}_\rho(k, a', m')\end{array}$$

Lemma 6 ($\sqsubseteq_{\rho,k}$ Reflexive and Transitive)
The extend-state relation ($\sqsubseteq_{\rho,k}$) is reflexive and transitive.

Consider a state (a, m) where all unallocated memory locations contain *junk*; that is, there are no "initialized but not yet allocated" locations. When we extend the state, $(a, m) \sqsubseteq_{\rho,k} (a', m')$, as a result of the **Heap Well-Typed** property of a valid state, we are forced to initialize a new memory location (with a value of the appropriate type) *before* we add it to the allocset:

Lemma 7 (Initialization Before Allocation)
$(a, m) \sqsubseteq_{\rho,k} (a', m') \;\Leftrightarrow\; (a, m) \sqsubseteq_{\rho,k} (a, m') \sqsubseteq_{\rho,k} (a', m')$

6 What is a type?

A type is a predicate on (i, k, a, m, x). We now describe the four properties a type must have in order to be considered a "good" or valid type.

Extensible. In section 3.3 we presented the **Update Invariance** rule which says that type judgments are preserved across memory extension. How can we use the stratified model of mutable references to prove this rule as a lemma? We start by stating update invariance as a property of a type-predicate, which says that a type (at level i) is closed under valid extension of the memory (i.e., state extension at level $i - 1$):

$$\begin{array}{l}\mathtt{update\text{-}inv}(\tau) \;=\\ \quad \forall i, k, x, a, m, m'.\;\; (a, m) \sqsubseteq_{\mathtt{rep}(i-1),k} (a, m') \\ \qquad\qquad\qquad \Rightarrow x :_{i,k,a,m} \tau \Rightarrow x :_{i,k,a,m'} \tau\end{array}$$

Notice that since $(a, m) \sqsubseteq_{\rho,k} (a, m')$ allows updates of both allocated and unallocated locations, `update-inv` incorporates the notion of **Initialization Invariance** described in section 3.2.

We model the allocation of new memory by extending the allocset. To reason about programs that dynamically allocate memory, we need a rule that says that type judgments are preserved under extension of the allocset. We call this the **Allocation Invariance** rule. In lieu of the rule we specify the allocation invariance property of a type-predicate:

$$\begin{array}{l}\mathtt{alloc\text{-}inv}(\tau) \;=\\ \quad \forall i, k, x, m, a, a'.\;\; (a, m) \sqsubseteq_{\mathtt{rep}(i-1),k} (a', m) \\ \qquad\qquad\qquad \Rightarrow x :_{i,k,a,m} \tau \Rightarrow x :_{i,k,a',m} \tau\end{array}$$

We say that a predicate τ on (i, k, a, m, x) is `extensible` if it has both the `update-inv` and `alloc-inv` properties which leads to the following definition:

$$\begin{array}{l}\mathtt{extensible}(\tau) \;=\\ \quad \forall i, k, x, a, m, a', m'.\;\; (a, m) \sqsubseteq_{\mathtt{rep}(i-1),k} (a', m') \\ \qquad\qquad\qquad \Rightarrow \tau(i, k, a, m, x) \Rightarrow \tau(i, k, a', m', x)\end{array}$$

If $\mathtt{extensible}(\tau)$ holds for each type τ in our system, then we can easily prove the **Update**, **Allocation** and **Initialization Invariance** rules as lemmas.

Index closed. A property of types that we have already mentioned is that if x has type τ for k future execution steps, then it must be the case that x has type τ for $j < k$ steps:

$$\begin{array}{l}\mathtt{kclosed}(\tau) \;=\\ \quad \forall i, k, j, a, m.\;\; 0 \le j < k \Rightarrow \tau(i, k, a, m, x) \Rightarrow \tau(i, j, a, m, x)\end{array}$$

Upward closed. Informally, a type at level i has more "information" than a type at level j, for $j < i$ — i.e., the former has access to more levels of `rep` and consequently, can determine the pretypes of more locations in the allocset. Therefore, if we have sufficient information at level i to conclude that x has type τ, then at level $i + 1$ we still have sufficient information to conclude that x has type τ:

$$\begin{array}{l}\mathtt{iclosed}(\tau) \;=\\ \quad \forall i, j, k, a, m.\;\; 0 \le i < j \Rightarrow \tau(i, k, a, m, x) \Rightarrow \tau(j, k, a, m, x)\end{array}$$

Representable. To construct a value of type τ, we would first write the value into an unallocated memory location l, and then extend the allocset with the pair (l, n) where n represents the pretype $\tau(i)$ at some level i in the Gödel numbering hierarchy. Clearly this last step requires the existence of such an i and n; i.e., there must be some $i \ge 0$ such that the pretype $\tau(i)$ is representable:

$$\mathtt{repable}(\tau, i) = \exists n.\, \mathtt{rep}(i, n, \tau(i))$$

Valid types. We say that a predicate τ on (i, k, a, m, x) is a `type` if it is extensible, index closed, upward closed and representable:

Definition 8 (Type)

$$\begin{array}{l}\mathtt{type}(\tau) \;=\; \mathtt{extensible}(\tau) \;\wedge\; \mathtt{kclosed}(\tau) \;\wedge\; \mathtt{iclosed}(\tau) \\ \qquad\qquad\quad \wedge\; \exists i.\, \mathtt{repable}(\tau, i)\end{array}$$

7 Modeling a nontrivial type system

In this section we present a model of general references. More precisely, our model permits references to values of any type defined using the primitive types and type contructors shown in figure 4; figure 3 gives the pretype definitions that the figure 4 definitions rely on. We will first specify a Gödel numbering relation for the pretypes in our system, then explain some of the more involved type definitions, and finally present the relevant theorems. The accompanying

$\boxed{\text{For } k \geq 0:}$

$$
\begin{array}{lcl}
\overline{\top}(k,a,m,x) & = & \texttt{true} \\
\overline{\bot}(k,a,m,x) & = & \texttt{false} \\
\overline{\mathsf{int}}(k,a,m,x) & = & \texttt{true} \\
(\overline{\mathsf{const}}(n))(k,a,m,x) & = & x = n \\
(\overline{\mathsf{offset}}(n,\sigma))(k,a,m,x) & = & \sigma(k,a,m,x+n) \\
(\sigma_1 \mathbin{\overline{\cup}} \sigma_2)(k,a,m,x) & = & \sigma_1(k,a,m,x) \\
 & & \lor\ \sigma_2(k,a,m,x) \\
(\sigma_1 \mathbin{\overline{\cap}} \sigma_2)(k,a,m,x) & = & \sigma_1(k,a,m,x) \\
 & & \land\ \sigma_2(k,a,m,x) \\
(\overline{\mathsf{rec}}\,F)(k,a,m,x) & = & F^{k+1}\bot(k,a,m,x) \\
(\overline{\mathsf{box}}(\rho,\sigma))(k,a,m,x) & = & \exists n.\,(x,n) \in a \\
 & & \land\ \rho(n, \mathbf{K}\,\overline{\mathsf{const}}(m(x))) \\
 & & \land\ \forall j < k.\, \sigma(j,a,m,m(x)) \\
(\overline{\mathsf{ref}}(\rho,\sigma))(k,a,m,x) & = & \exists n.\,(x,n) \in a \\
 & & \land\ \rho(n, \mathbf{K}\,\sigma_n)\ \land\ \sigma_n = \sigma \\
 & & \land\ \forall j < k.\, \sigma(j,a,m,m(x)) \\
(\overline{\mathsf{codeptr}}(\rho,\sigma))(k,a,m,x) & = & \forall r',a',m',j.\ j < k \\
 & & \ \land\ r'(\textsc{pc}) = x \\
 & & \ \land\ (a,m) \sqsubseteq_{\rho,j} (a',m') \\
 & & \ \land\ \sigma(j,a',m',r'(1)) \\
 & & \Rightarrow \texttt{safen}(j,r',m')
\end{array}
$$

Figure 3. Pretype definitions

technical report [3] contains a more detailed exposition of our model (including existential and universal types which we have not described here due to lack of space), as well as proofs of the theorems in this paper.

7.1 Godel numbers of pretype functions

Our model requires that we specify the Gödel number of each of the pretypes in figure 3. But to specify the Gödel number of $\overline{\mathsf{rec}}(F)$ (where F is a function from pretypes to pretypes), we have to first specify the Gödel numbers of pretype functions F. Rather than extend the `rep` relation so that it specifies the Gödel numbers of pretype functions as well as of pretypes, we observe that if we represent every pretype as a pretype function that simply ignores its argument, then we would only require Gödel numbers of pretype functions. We use $\mathbf{K}\,\sigma$ to denote a pretype function that ignores its argument and returns the pretype σ (i.e., $\mathbf{K} = \lambda y.\,\lambda x.\,y$).

We define $\texttt{rep}(i,n,F)$, the Gödel numbering relation for pretype functions, in the same way as we defined $\texttt{rep}(i,n,\sigma)$ in section 4.3. We present some of the rules characterizing the relation in figure 5. Note that `base` now consists of the Gödel numbers of $\mathbf{K}\,\sigma$ for primitive pretypes σ, as well as, the Gödel number of the identity pretype function $\lambda\sigma.\,\sigma$. For `closure`, if the pretype functions F and the pretype σ are representable at level i, then $F(\sigma)$ should also be representable at level i. (This condition is specified by the $\mathsf{clos\_apply}$ rule in figure 5.) In addition, the pretype constructors $\overline{\mathsf{offset}}$, $\overline{\cup}$, $\overline{\cap}$, and $\overline{\mathsf{rec}}$ are used to compute the `closure` of each ρ where $\rho \subset \texttt{rep}(i)$. Given some $\rho = \texttt{rep}(i)$, the pretypes $\overline{\mathsf{box}}$, $\overline{\mathsf{ref}}$, and $\overline{\mathsf{codeptr}}$, (each of which is parametrized by ρ) are used to compute $\texttt{step}(\rho)$. For example, if F is representable at level i, then the pretype functions $\lambda\sigma.\,\overline{\mathsf{ref}}(\texttt{rep}(i), F(\sigma))$ and $\mathbf{K}\,\overline{\mathsf{rec}}(\lambda\sigma.\,\overline{\mathsf{ref}}(\texttt{rep}(i), F(\sigma)))$ are representable at level $i+1$. We modify the definitions of `godel`, `match`, and `repable` as follows:

$$
\begin{array}{lcl}
\texttt{godel}_\rho(n) & = & \exists \rho',\sigma.\ \rho \subset \rho'\ \land\ \rho'(n, \mathbf{K}\,\sigma) \\
\texttt{match}_\rho(k,n,a,m,x) & = & \forall \rho',\sigma.\ \rho \subset \rho'\ \Rightarrow\ \rho'(n,\mathbf{K}\,\sigma) \\
 & & \qquad \Rightarrow\ \sigma(k,a,m,x) \\
\texttt{repable}(F,i) & = & \exists n.\,\texttt{rep}(i,n,\overline{F(i)})
\end{array}
$$

$\boxed{\text{For } i \geq 0:}$

$$
\begin{array}{lcl}
\top(i) & = & \overline{\top} \\
\bot(i) & = & \overline{\bot} \\
\mathsf{int}(i) & = & \overline{\mathsf{int}} \\
(\mathsf{const}(n))(i) & = & \overline{\mathsf{const}}(n) \\
(\mathsf{offset}(n,\tau))(i) & = & \overline{\mathsf{offset}}(n,\tau(i)) \\
(\tau_1 \cup \tau_2)(i) & = & \tau_1(i) \mathbin{\overline{\cup}} \tau_2(i) \\
(\tau_1 \cap \tau_2)(i) & = & \tau_1(i) \mathbin{\overline{\cap}} \tau_2(i) \\
(\mathsf{rec}\ F)(i) & = & \overline{\mathsf{rec}}(\overline{F(i)})
\end{array}
$$

$\boxed{\text{For } i > 0:}$

$$
\begin{array}{lcl}
(\mathsf{box}\ \tau)(i) & = & \overline{\mathsf{box}}(\texttt{rep}(i-1), \tau(i-1)) \\
(\mathsf{ref}\ \tau)(i) & = & \overline{\mathsf{ref}}(\texttt{rep}(i-1), \tau(i-1)) \\
(\mathsf{codeptr}\ \tau)(i) & = & \overline{\mathsf{codeptr}}(\texttt{rep}(i-1), \tau(i-1))
\end{array}
$$

where, $\overline{F(i)} : \textit{pretype} \rightarrow \textit{pretype}\ =\ \lambda\tau.\,(F(\lambda i'.\tau))(i)$

Figure 4. Type definitions

7.2 Types

Recursive types (rec**).** We have incorporated Appel and McAllester's [5] indexed model of types where the indices k allow the construction of a well-founded recursion, even when modeling contravariant recursive types. All definitions and proofs regarding recursive types are explained in their paper. In proving that a type $\mathsf{rec}(F)$ is a valid type (theorem 11), we require that F be `wellfounded` (see [5]).

Immutable references (box**).** The type box of immutable references is defined in figures 3 and 4. Any value may be stored in an immutable location, as long as it is numerically equal to the value that is already there.

First-order continuations ($\mathsf{codeptr}$**).** A value of type $\mathsf{codeptr}(\tau)$ is a first-order continuation, that is, a machine-code address to which control may be passed, provided that its precondition (that register $\mathbf{r}_1$ contain a value of type τ) is satisfied. Consider the judgment $x :_{i,k,a,m} \mathsf{codeptr}(\tau)$.

$$\frac{}{\mathtt{rep}(0, \mathtt{tree}_0(1), \mathbf{K}\,\overline{\top})}\ \textsf{base\_}\top \qquad \frac{}{\mathtt{rep}(0, \mathtt{tree}_1(4, \mathtt{tree}_0(c)), \mathbf{K}\,\overline{\mathsf{const}}(c))}\ \textsf{base\_const} \qquad \frac{}{\mathtt{rep}(0, \mathtt{tree}_0(7), \lambda\tau.\,\tau)}\ \textsf{base\_id}$$

$$\frac{\mathtt{rep}(i, n_1, F) \qquad \mathtt{rep}(i, n_2, \mathbf{K}\,\tau)}{\mathtt{rep}(i, \mathtt{tree}_2(11, n_1, n_2), \mathbf{K}\,(F(\tau)))}\ \textsf{clos\_apply} \qquad \frac{\mathtt{rep}(i, n, F)}{\mathtt{rep}(i, \mathtt{tree}_2(12, \mathtt{tree}_0(c), n), \lambda\tau.\,\overline{\mathsf{offset}}(c, F(\tau)))}\ \textsf{clos\_offset}$$

$$\frac{\mathtt{rep}(i, n_1, F_1) \qquad \mathtt{rep}(i, n_2, F_2)}{\mathtt{rep}(i, \mathtt{tree}_2(13, n_1, n_2), \lambda\tau.\,F_1(\tau)\,\overline{\cup}\,F_2(\tau))}\ \textsf{clos\_}\cup \qquad \frac{\mathtt{wellfounded}(F) \qquad \mathtt{rep}(i, n, F)}{\mathtt{rep}(i, \mathtt{tree}_1(15, n), \mathbf{K}\,\overline{\mathsf{rec}}(F))}\ \textsf{clos\_rec}$$

$$\frac{\mathtt{rep}(i, n, F)}{\mathtt{rep}(i+1, n, F)}\ \textsf{step\_include} \qquad \frac{\mathtt{rep}(i, n, F)}{\mathtt{rep}(i+1, \mathtt{tree}_2(22, \mathtt{tree}_0(i), n), \lambda\tau.\,\overline{\mathsf{ref}}(\mathtt{rep}(i), F(\tau)))}\ \textsf{step\_ref}$$

Figure 5. Some clauses in the Godel numbering relation for pretype functions

This says that if we jump to address x in some future machine state (r', m') (i.e., if $r'(\text{PC}) = x$), then (r', m') is a safe state for $j < k$ steps if the following three conditions hold. First, the argument (i.e., the value stored in register $\mathbf{r}_1$) must be of the right type; that is, $r'(1) :_{i-1,j,a',m'} \tau$ (see codeptr in figure 3). A crucial observation about $\mathsf{codeptr}(\rho, \sigma)$ is that if $\rho = \mathtt{rep}(i')$, then σ is a pretype at level i'.

Second, state (a', m') must be a valid extension of state (a, m) — i.e., $(a, m) \sqsubseteq_{\rho,j} (a', m')$. Note that in extending (a, m) to (a', m') we may have created new cells with types that are higher in the Gödel numbering hierarchy. In that event we need a higher level `rep` relation, say ρ' such that $\rho \subset \rho'$, to ensure that state (a', m') is valid. But recall the definition of `validstate`, or more specifically, the definitions of `godel` and `match` in section 5.1. The predicates `godel` and `match` are defined in such a way that we have $\mathtt{validstate}_\rho(j, a', m')$ if and only if $\mathtt{validstate}_{\rho'}(j, a', m')$ for $\rho \subset \rho'$ (see lemma 10).

Third, the program p that is in memory in state (a, m) must still be in memory in state (a', m'). To enforce this condition, we make all allocset locations that contain program instructions “immutable”; that is, the allocset maps these locations to Gödel numbers of box types. Since the program is preserved under state extension, our requirement is simply $(a, m) \sqsubseteq_{\rho,j} (a', m')$.

7.3 Validity of types

Lemma 9 (`rep` Upward Closed)
$\mathtt{rep}(i, n, F) \Rightarrow \mathtt{rep}(i+1, n, F)$

Lemma 10 (State Valid Modulo Rep Level)
If $\mathtt{rep}(i)$ *is a sub-relation of* $\mathtt{rep}(i')$ *then:*
$\mathtt{validstate}_{\mathtt{rep}(i)}(k, a, m) \Leftrightarrow \mathtt{validstate}_{\mathtt{rep}(i')}(k, a, m)$

The above lemma is *key* to the consistency of our model. It follows from lemma 9 and our carefully crafted definitions of `godel` and `match`, and allows us to prove a series of lemmas critical to the proof of the following theorem [3].

Theorem 11 (Type)
a-d. $\mathtt{type}(\tau)$, *where* $\tau ::= \top | \bot | \mathsf{int} | \mathsf{const}(n)$
e. $\mathtt{type}(\tau) \Rightarrow \mathtt{type}(\mathsf{offset}(i, \tau))$.
f. $\mathtt{type}(\tau_1) \wedge \mathtt{type}(\tau_2) \Rightarrow \mathtt{type}(\tau_1 \cup \tau_2)$.
g. $\mathtt{type}(\tau_1) \wedge \mathtt{type}(\tau_2) \Rightarrow \mathtt{type}(\tau_1 \cap \tau_2)$.
h. $\mathtt{wellfounded}(F) \Rightarrow \mathtt{type}(\mathsf{rec}(F))$.
i. $\mathtt{type}(\tau) \Rightarrow \mathtt{type}(\mathsf{box}\,\tau)$.
j. $\mathtt{type}(\tau) \Rightarrow \mathtt{type}(\mathsf{ref}\,\tau)$.
k. $\mathtt{type}(\tau) \Rightarrow \mathtt{type}(\mathsf{codeptr}\,\tau)$.

We can construct a variety of types using the types shown in figure 4 (and $\exists$, $\forall$ types [3]) as building blocks (see Appel and Felty [4]). Any such type (except arbitrarily nested recursive and quantified types — see section 9) can easily be shown to be a `type`.

7.4 Invariance lemmas

Lemma 12 (Initialization Invariance)

$$\frac{x :_{i,k,a,m} \tau \qquad y \notin a \qquad m' = m\,[y := z] \qquad \mathtt{type}(\tau)}{x :_{i,k-1,a,m'} \tau}$$

Lemma 13 (Allocation Invariance)

$$\frac{\begin{array}{c} x :_{i,k,a,m} \tau \qquad a' = a \cup \{(y, n)\} \\ (y \notin a \ \vee \ (y, n) \in a) \qquad \mathtt{type}(\tau) \end{array}}{x :_{i,k,a',m} \tau}$$

Lemma 14 (Update Invariance)

$$\frac{\begin{array}{c} x :_{i,k,a,m} \tau \qquad y :_{i,k,a,m} \mathsf{ref}\ \tau' \qquad z :_{i,k,a,m} \tau' \\ m' = m\,[y := z] \qquad \mathtt{type}(\tau) \end{array}}{x :_{i,k-1,a,m'} \tau}$$

We can show that the premises of the above lemmas satisfy the extend-state ($\sqsubseteq$) relation. The conclusions then follow trivially from $\mathtt{type}(\tau)$.

7.5 Introduction & elimination rules

We prove introduction and elimination lemmas for each type in figure 4 [3]. We only show the lemmas for ref here.

Lemma 15 (Ref-I)

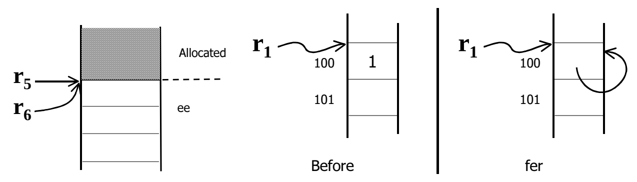
$$\frac{x \notin a \qquad m(x) :_{i,k,a,m} \tau \qquad \texttt{repable}(\tau, i) \qquad \texttt{type}(\tau)}{\exists a'.\, (a, m) \sqsubseteq_{\texttt{rep}(i),k} (a', m) \;\wedge\; x :_{i+1,k+1,a',m} \mathsf{ref}\ \tau}$$

Lemma 16 (Ref-E)

$$\frac{x :_{i+1,k+1,a,m} \mathsf{ref}\ \tau}{m(x) :_{i,k,a,m} \tau}$$

8 Two examples

Using our model of general references we can prove lemma 2 for the following program fragments. The proofs are given in the technical report [3].



(a) Before example 1 (b) Before and after example 2

Figure 6. Scenarios for examples

Example 1 (Initialize, allocate, update)
Consider figure 6(a) as the starting point of the following program fragment at address l in memory, which constructs a reference cell of type τ_2 (i.e., initializes a location, then adds it to the allocset) and then updates it. Note that τ_2 may be any type definable in our system. For this example, assume that register $\mathbf{r_6}$ is a special register that always points to the next address to be allocated; then, to add a location to the allocated set, we must simply increment this register. For brevity, we have omitted "$\lambda k, r, m.$" from the following invariants.

$\{\exists a, i.\ r(1) :_{i,k,a,m} \tau_1 \wedge r(4) :_{i,k,a,m} \tau_2$
$\quad \wedge\, r(5) \notin \texttt{dom}(a) \wedge r(5) = r(6)\}$
$\mathbf{m(r_5) \leftarrow r_4}$
$\{\exists a, i.\ r(1) :_{i,k,a,m} \tau_1 \wedge r(4) :_{i,k,a,m} \tau_2 \wedge m(r(5)) :_{i,k,a,m} \tau_2\}$
$\mathbf{r_6 \leftarrow r_6 + 1}$
$\{\exists a, i.\ r(1) :_{i,k,a,m} \tau_1 \wedge r(4) :_{i,k,a,m} \tau_2 \wedge r(5) :_{i,k,a,m} \mathsf{ref}\ \tau_2\}$
$\mathbf{m(r_5) \leftarrow r_4}$
$\{\exists a, i.\ r(1) :_{i,k,a,m} \tau_1 \wedge r(5) :_{i,k,a,m} \mathsf{ref}\ \tau_2\}$

Example 2 (Cycles in memory)
Our model can handle cycles in the memory graph. Figure 6(b) depicts the situation before and after executing the store instruction in the following Hoare triple, where F denotes $\lambda\tau.\ \mathsf{ref\ int} \cup \mathsf{ref}\ \tau$:

$\{\exists a, i.\ r(1) :_{i,k,a,m} \mathsf{ref\ int} \wedge 100 :_{i,k,a,m} \mathsf{rec}(F)\}$
$\mathbf{m(r_1) \leftarrow r_1}$
$\{\exists a, i.\ 100 :_{i,k,a,m} \mathsf{rec}(F)\}$

9 Typed machine language

A limitation of the semantics we described in section 7 is that it does not allow us to represent arbitrarily nested recursive and quantified types such as the following:

$$\mathsf{rec}(\lambda\alpha.\ \mathsf{ref}(\mathsf{rec}(\lambda\beta.\ (\mathsf{ref}\ \beta) \cup \alpha)))$$

Typed Machine Language, described by Swadi and Appel, [31] accommodates arbitrarily nested recursive and quantified types using DeBruijn indices. Our new semantics is compatible with this approach.

10 Machine-checked proof

All of our proofs are machine-checked, and furthermore, these proofs have an actual use: they form part of the proof of safety of a machine-language program in a PCC system. The logic that we use is Church's higher-order logic with a few axioms of arithmetic; we represent our logic, and check proofs, in the LF metalogic [10] implemented in the Twelf logical framework [25]. Our proof "implementation" consists of about 16,500 lines of Twelf code, using the encoding of higher-order logic described by Appel and Felty [4]. The implementation of the Appel-McAllester model (without mutable references) consisted of 8,200 lines of proof, while that of the Appel-Felty model (with neither contravariant recursive types nor mutable references) consisted of 5,400 lines.

11 Conclusions & related work

Denotational vs. structured operational semantics. In our model we prove all type inference rules as derived lemmas. The reader may have wondered if our denotational semantics approach or an operational (purely syntactic) approach is the right one to take. We should observe that our model is a mix of denotional and operational semantics. Our `validstate` predicate behaves like the induction hypothesis of a syntactic progress-and-preservation proof, though in the indexed model the induction is over the number of steps that can be executed in the future rather than the past. We have used the semantic approach to build a syntactic type system for Sparc instructions with 1032 clauses (including both operator definitions and typing rules). The syntactic proof of the soundness of this type system would require the analysis of close to a thousand cases for each property proved by induction. A person doing the syntactic proof of this theorem would, no doubt, tire of the large number of cases and soon settle on an über-property to use as the induction hypothesis, just as we have done using `validstate`. There really is a continuum between the purely syntactic and purely semantic approaches — types in the Appel-Felty [4] model were sets of terms, in Appel-McAllester [5] they were domains of terms, and in this model we have domains of terms with embedded syntax. We believe that in practice our approach has turned out to be more modular than a purely syntactic approach.

Compositionality and syntax-in-the-semantics. We have presented a model of general references based on possible-worlds semantics. This model is not compositional in the conventional sense. To see why, consider the fact that a value x is of type $\mathsf{ref}\ \tau$ if and only if it has type τ in the extensional sense (which, in our model, means by inspection of the registers and memory) and *also* has type τ in the intensional sense. The intension indicates that x will continue to be of type τ indefinitely. References in ML and Java have an intensional component: when a ref cell is allocated it is tagged with the type that it must have for its lifetime. We suspect that only purely extensional concepts can have a (strictly) compositional semantics. Kripkean possible worlds have proven extremely valuable in modeling intension. So let us consider the nature of possible worlds. Peregrin's [24] analysis concludes that "a possible world in the intuitive sense can be explicated as a maximal consistent class of statements". This implies that to give the semantics of possible worlds we require techniques like coinduction [23, 17, 6] or non-well-founded sets [2], each of which is in some sense syntax-dependent. In light of that, and Peregrin's conclusion that "possible worlds are language dependent", our embedding of syntactic types (that express intension) in the semantics seems unavoidable. But the latter should not be interpreted as: "the use of Gödel numbers in the semantics seems unavoidable" — we use Gödel numbers to encode our stratified semantics in higher-order logic, but even without that, the stratified semantics itself contains syntactic types (in the $allocset_i$).

Game semantics. Hintikka [11] advocated the use of game-theoretical semantics to model possible worlds. Game semantics is especially useful in removing from consideration all *impossible* possible worlds. Abramsky, Honda and McCusker[1] describe a game semantics of general references that they show to be fully abstract. In this model, reference types are modeled by their behavior, or more specifically as a product of a "read method" and a "write method" in the style of Reynolds [27]. This representation is quite different from that in location-based models such as ours. It would be interesting to see if such a model could be incorporated into a foundational PCC system.

Levy's possible-worlds model. Recently, Levy [15] described a possible-worlds model for general references. There are interesting correspondences between his model and ours. His *world-store* (w, s), where w is a world and s is a w-store, (i.e., each location in s is well-typed with respect to w) corresponds to a valid state (a, m) in our model. His accessibility relation between worlds resembles ours. His model has the property: "if $w \leq w'$ then every w_τ-value is a w'_τ-value" (where w_τ-value denotes a value of type τ in world w); this corresponds to `extensible` in our model. The definition (denotation to be more precise) of the type $\mathsf{ref}\ \tau$ in his model is: $[\![\mathsf{ref}\ \tau]\!]w = \$w_\tau$ (where $\$w_\tau$ denotes "the set of cells of type τ in w"). Notice that $[\![\mathsf{ref}\ \tau]\!]$ is defined in terms of the syntax τ rather than the semantics $[\![\tau]\!]$; that is, this semantics is not compositional either. Levy is faced with the same kind of circularity that we described in section 4. He solves it by observing that recursive equations on domains have solutions. We solve it by showing that our hierarchy of types has a limit.

Locality of reasoning. We have shown how our semantic model provides us with rules (lemmas) that allow us to prove properties of programs with mutable references — as long as these properties are expressed as types. There has been a great deal of work on program-proving for pointers; here, we discuss only the formalism described by Ishtiaq and O'Hearn [14] (which is closely related to the work of Burstall and Reynolds [7, 28]). When proving properties of programs that mutate the heap, a great deal of effort is spent reasoning about what does *not* change. Ishtiaq and O'Hearn use the BI logic [20] which provides a spatial form of conjunction $*$ such that the statement $P * Q$ is true just when the current heap can be split into two components, one of which makes P true and the other of which makes Q true. This operator allows them to introduce frame axioms (which describe invariants of the heap) using the rule,

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\ \mathit{ModifiesOnly}(C) \cap \mathit{free}(R) = \emptyset$$

where $\mathit{ModifiesOnly}(C)$ is the set of (free) variables that are updated by the command C. This resembles the following rule in our system, though in our model, R is restricted to type judgments such as $r(1) :_{a,m} \tau_1 \ \wedge 100 :_{a,m} \tau_2$ (where 100 may be a memory location). Also, in our model, $\mathit{ModifiesOnly}(C)$ is the set of *registers* that are updated by the command C and $\mathit{free}(R)$ is the set of registers about which R contains assertions:

$$\frac{\{P\}C\{Q\}}{\{P \wedge R\}C\{Q \wedge R\}}\ \mathit{ModifiesOnly}(C) \cap \mathit{free}(R) = \emptyset$$

The use of $\wedge$ instead of $*$ has important consequences. Consider the situation in figure 1(b). Now suppose we have the following C, P, Q, and R:

$$\begin{array}{l} P = \exists a.\ r(5) :_{k,a,m} \mathsf{ref}\ \tau_2 \wedge r(4) :_{k,a,m} \tau_2 \\ C = \mathbf{m(r_5)} \leftarrow \mathbf{r_4} \\ Q = \exists a.\ r(5) :_{k,a,m} \mathsf{ref}\ \tau_2 \\ R = \exists a.\ r(1) :_{k,a,m} \tau_1 \end{array}$$

If $\{P\}C\{Q\}$ holds, then using our frame-axiom introduction rule, we can conclude $\{P \wedge R\}C\{Q \wedge R\}$ holds. We cannot conclude $\{P * R\}C\{Q * R\}$ using their frame-axiom introduction rule because in this situation (figure 1(b)) the heap cannot be split into two parts, such that one part satisfies $r(5) :_{a,m} \mathsf{ref}\ \tau$ and the other part $r(1) :_{a,m} \tau_1$. Their frame axiom introduction rule is useful when aliasing is not expected to occur because their predicates R are stronger than just typing judgments.

Other related work. The use of a store typing mapping locations to types is not new. Tofte [32] uses this approach in his type soundness proof for polymorphic references. Tofte, however, makes use of coinduction to handle cycles in the memory graph. Harper [9] has shown how a progress-and-preservation proof can be arranged so that there is no need for coinduction. Our model, meanwhile, can handle cycles in memory by virtue of the index k. For a reference to a memory cycle to be well-typed, we only need to know that it is well-typed to approximation k. With each memory dereference the k decreases. Hence, there is no need for coinduction.

A common feature of models of mutable state is that they specify *how* the state is allowed to vary over time. Models for *Idealized Algol* developed by Reynolds and Oles [27, 21, 22] make use of functor categories indexed by possible worlds or *store shapes* to specify how the size of the store, as well as its contents, may change at any point in the program. We note, however, that these models do not handle general references. Stark [30] (building on work done with Pitts on possible world models of the nu-calculus [26]) describes a denotational semantics for *Reduced ML* that includes integer references.

12 Future work

All practical languages provide some means for managing memory, but the model that we have described does not allow memory to be reclaimed. Building a possible-worlds model that permits deallocation is challenging in the context of ML-style references where the type of a memory cell is fixed upon allocation; (technically, the difficulty lies in defining an extend-state relation that permits deallocation *and* is transitive). We are attempting to extend our possible-worlds model to allow region-based memory management.

Appel and McAllester's indexed model [5] has both a simple, nonextensional version and an extensional version using PERs. It is not trivial to make an extensional semantics for general references because the equivalence of two values depends on the set of their free locations. We intend to investigate this.

Acknowledgements. We would like to thank Peter O'Hearn for noting that our model is a possible-worlds model and for pointing out Paul Levy's related work, and the anonymous referees for various helpful comments and suggestions.

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] P. Aczel. *Non-Well-Founded Sets*. Center for the Study of Language and Information, Stanford University, 1988.

[3] A. J. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. Technical Report TR-650-02, Princeton University, May 2002.

[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. Technical Report TR-629-00, Princeton University, Oct. 2000.

[6] J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena.* Cambridge University Press, 1996.

[7] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[8] 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/`.

[9] 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.

[10] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[11] K. J. Hintikka. Impossible possible worlds vindicated. *Journal of Philosophical Logic*, 4:475–484, 1975.

[12] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[13] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, Cambridge, England, 2000.

[14] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001. ACM.

[15] P. B. Levy. *Call-by-Push-Value*. Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001.

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

[17] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[18] 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.

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

[20] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[21] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph. D. dissertation, Syracuse University, Syracuse, New York, August 1982.

[22] F. J. Oles. Functor categories and store shapes. In P. W. O'Hearn and R. D. Tennent, editors, *ALGOL-like Languages, Volume 2*, pages 3–12. Birkhäuser, Boston, Massachusetts, 1997.

[23] D. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 59–78, Edinburgh, 1969. Edinburgh University Press.

[24] J. Peregrin. Possible worlds: A critical analysis. *The Prague Bulletin of Mathematical Linguistics*, 59-60:9–21, 1993.

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

[26] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In A. M. Borzyszkowski and S. Sokołowski, editors, *Mathematical Foundations of Computer Science 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, 1993. Springer-Verlag.

[27] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[28] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[29] S.A. Kripke. Semantical considerations on modal logic. In *Proceedings of a Colloquium: Modal and Many Valued Logics*, volume 16, pages 83–94, 1963.

[30] I. D. B. Stark. *Names and Higher-Order Functions*. Ph. D. dissertation, University of Cambridge, Cambridge, England, December 1994.

[31] K. N. Swadi and A. W. Appel. Foundational semantics for tal syntactic rules via typed machine language. Submitted for publication, 2002.

[32] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.