Semantics of General References by a Hierarchy of Gödel Numberings

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. 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 [13, 12], 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 [2], 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 [2] have shown how to extend this to allocation of immutable values, covariant recursive types, function pointers, and quantified types. Appel and McAllester [3] further extend this to contravariant recursive types. With each extension, the semantic model gets more complicated. 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. The semantic model contains an interesting use of Gödel numbering as a way to model denotational semantics. After all, denotational semantics and Gödel numbering address the same problem, which is the relation of syntax to semantics.

Almost [8] 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. 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 the these rules; for this we need a semantic model.

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.

2. Foundational Proofs of Safety

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

2.1 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 the 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 *readable* addresses will be loaded," (where the predicate `readable` is suitably specified as part of the safety policy), we can define the instruction $\mathbf{r_i} \leftarrow \mathbf{m}[\mathbf{r_j} + \mathbf{c}]$ as:

$$\begin{aligned}&\texttt{load}(i, j, c) \;=\\ &\quad \lambda r, m, r', m'.\ \texttt{readable}(r(j)+c) \wedge r'(i) = m(r(j)+c)\\ &\qquad\qquad \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m\end{aligned}$$

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 $r(\mathrm{PC})$ [1] 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'):

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

[1] PC denotes the program counter.

Preliminary version, July 2001

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

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 `safe-state`(r,m)) amounts to showing that there is no path from (r,m) to a state with no successor. To prove a program safe we just have 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.

2.2 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. In foundational PCC (as in Necula [13]) these preconditions are expressed using types, e.g., $r(1) :_m \tau_1 \wedge r(5) :_m \tau_5$. A judgment $x :_m \tau$ may be read as "x has type τ with respect to memory m."

We present three examples of program fragments that read from memory, initialize memory and update memory, respectively, and show type-inference rules required to prove these fragments safe. In a foundational system these inference rules cannot be added to the logic as axioms; we explain how they can be proved as lemmas.

Example 1 (Traversal of Heap-Allocated Data)
Consider the following program fragment which reads a value from a data structure in memory. The precondition — i.e., the invariant at address 101, written $I_{101}(r,m)$ — says that this data structure must be a reference cell containing a value of type τ in memory m. The postcondition, i.e., the invariant $I_{102}(r,m)$, requires that the value in the destination register have type τ with respect to memory m.

$$\begin{array}{rl} & I_{101}(r,m) = r(2) :_m \textbf{ref}\ \tau \\ 101: & \mathbf{r_3} \leftarrow \mathbf{m}(\mathbf{r_2}) \\ & I_{102}(r,m) = r(3) :_m \tau \end{array}$$

In proving a program safe, for each program point l we must prove `progress` and `preservation`. The property `progress`(l) says that if the invariant at address l holds, and if there is a valid instruction at address l, then we can safely execute this instruction:

$$\begin{array}{l} \texttt{progress}(l) = \\ \quad \forall r,m,i.\ r(\text{PC}) = l \wedge I_l(r,m) \wedge \texttt{decode}(m(l),i) \\ \qquad \Rightarrow\ \exists r',m'.\ (r,m) \mapsto (r',m') \end{array}$$

We will not discuss proofs of `progress` here [2] — in this paper we are mainly concerned with proving `preservation`. The property `preservation`(l) 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 $I_{r(PC)}(r,m)$ is satisfied. More formally, we can say that we have to establish statements of the form $\{P\}C\{Q\}$ where C is an instruction (or command) and P and Q are the pre- and postconditions, respectively. If we ignore control-flow instructions (which, in any case, are irrelevant to the material in this paper) [3] `preservation`(l) can be written as $\{I_l\}C\{I_{l+1}\}$ where C is the instruction stored at location l:

$$\begin{array}{l} \{I_l\}C\{I_{l+1}\} = \\ \quad \forall r,m,r',m'.\ r(\text{PC}) = l \wedge \texttt{decode}(m(l),C) \\ \qquad\qquad \wedge\ I_l(r,m) \wedge (r,m) \mapsto (r',m') \\ \qquad\ \Rightarrow\ r'(\text{PC}) = l+1 \wedge I_{l+1}(r',m') \end{array}$$

Then, for the above program fragment we have to show

$$\{I_{101}\}\mathbf{r_3} \leftarrow \mathbf{m}(\mathbf{r_2})\{I_{102}\}$$

The step relation increments the program counter, so $r'(\text{PC}) = 102$ is easily proved. Since we know from the semantics of the load instruction that $m' = m$, to prove $r_3 :_{m'} \tau$, 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, then we may conclude that the contents of memory at address x are of type τ:
Ref Elimination

$$\frac{x :_m \textbf{ref}\ \tau}{m(x) :_m \tau}$$

But where does this inference rule come from? A *type-specialized* PCC system (such as Necula's [13]) would include the above 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. Consider a model of types as sets of values, where 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). We then define types as predicates on values so that the judgment $x :_m \tau$ is just syntactic sugar for $\tau(m,x)$. For example, we can define integer and reference types as,

$$\begin{array}{lcl} \textbf{int}(m,x) & = & \text{true} \\ \textbf{ref}\ \tau(m,x) & = & \texttt{readable}(x) \wedge \tau(m,m(x)) \end{array}$$

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, [4] and using them as building blocks to describe more complicated datatypes, we can reason about the safety of programs that traverse nontrivial data structures — as long as these data structures are statically allocated.

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 [2] describe a semantic model that allows us to reason about the safety of such programs. Consider the following program fragment (which we write directly as a Hoare triple) that creates a new reference cell in memory by writing to a "new" memory location — register r_5 points to this new location:

$$\begin{array}{l} \{I_{111} = \lambda r,m.\ r(1) :_m \tau_1 \wedge r(4) :_m \tau_2\} \\ \mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4} \\ \{I_{112} = \lambda r,m.\ r(1) :_m \tau_1 \wedge m(r(5)) :_m \tau_2\} \end{array}$$

Given $r_1 :_m \tau_1$ and the fact that the store instruction alters memory (i.e., $m' \neq m$), how can we prove $r_1 :_{m'} \tau_1$? Consider the scenario illustrated by figure 1 — the store instruction updates the location that r_5 points to (thereby modifying the data structure that r_1

[2] To prove progress we essentially need to show that $I_l(r,m)$ satisfies the preconditions specified by instruction i. For example, the instruction `load`$(2,1,0)$ specifies the precondition `readable`$(r(2))$ — this condition must be part of the invariant $I_l(r,m)$. Proofs of progress require that we have the right invariants.

[3] While we have chosen to ignore control-flow instructions to simplify the presentation, our system can handle such instructions (see Appel and McAllester [3]).

[4] Appel and Felty [2] provide an extensive catalog of type constructors defined as predicates on values.

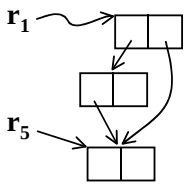


Figure 1: Pointer Aliasing

points to), so that we cannot know if r_1 has type τ_1 with respect to the modified memory m'. But the situation in figure 1 cannot occur since we stipulated that r_5 points to a "new" memory location.

If we distinguish between allocated and unallocated memory and ensure that only unallocated locations can be modified, then we can prove that type judgments are preserved across memory updates. For this, we maintain a set a of allocated addresses and say that a *state* is a pair (a,m) of an allocset a [5] and a memory m. Then, a value is a pair $((a,m),x)$ of a state and a root-pointer and types are, as before, predicates on values. Only `readable` and `writable` location are added to the allocset. The types **int** and **ref** are defined as, [6]

$$\begin{array}{lcl} \mathbf{int}(a,m,x) & = & \texttt{true} \\ \mathbf{ref}\ \tau(a,m,x) & = & x \in a \wedge \tau(a,m,m(x)) \end{array}$$

In this model, we can prove the following *Initialization Invariance* rule as a lemma. This 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

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

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

$$\begin{array}{l} \{I_{111} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(5) \notin a \wedge r(4) :_{a,m} \tau_2\} \\ \mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4} \\ \{I_{112} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge m(r(5)) :_{a,m} \tau_2\} \end{array}$$

Example 3 (Mutable Data Structures)
The model described by Appel and Felty [2] 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:

$$\begin{array}{l} \{I_{121} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(6) :_{a,m} \mathbf{ref}\ \tau_2 \\ \qquad\qquad \wedge r(4) :_{a,m} \tau_2\} \\ \mathbf{m}(\mathbf{r_6}) \leftarrow \mathbf{r_4} \\ \{I_{122} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(6) :_{a,m} \mathbf{ref}\ \tau_2\} \end{array}$$

To prove the above statement we need the *Update Invariance* rule. 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

$$\frac{x :_{a,m} \tau \qquad y :_{a,m} \mathbf{ref}\ \tau' \qquad z :_{a,m} \tau' \qquad m' = m[y:=z]}{x :_{a,m'} \tau}$$

For foundational PCC, we must devise a 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.

3. Modeling Permissible Heap Updates

In the semantics of immutable fields described by Appel and Felty [2] a type is a predicate on 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,

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

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

3.1 1st Flawed Model: Types and the Extended Allocset

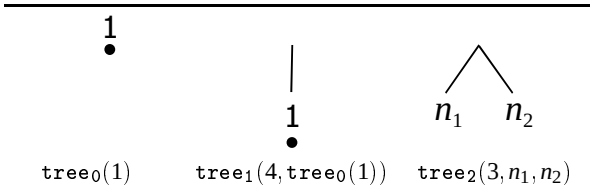
To allow for the update of existing values we enhance the allocset a to become a finite map from locations to types: for each allocated address x, we keep track of the type τ of updates allowed at x. As before, a type is a predicate on three arguments (a,m,x):

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

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. To eliminate the circularity in the definition of *type*, we present a solution that replaces the type (in the allocset) with the Gödel number of a type.

3.2 2nd Flawed Model: Gödel Numbers of Types

To describe the Gödel number of a type, we specify the relation $\texttt{rep}(n,\tau)$ which says that the Gödel number n represents the type τ. Rather than use integers to represent types, we opted to use finite trees of integers. A tree constructor $\texttt{tree}_i(c_0,t_1,\ldots,t_i)$ returns a tree with integer c_0 at the root and i subtrees $t_1,\ldots,t_i$, for example:



[5] The allocset a is virtual: it is existentially quantified and in a machine state (r,m), it can be computed by the program. The `alloc` predicate specifies the allocset a: $a = \texttt{alloc}(r,m)$. Note that the allocset must be computable by the program itself so that it knows where to find a location for new heap cells.

[6] For readability, we write $\mathbf{int}(a,m,x)$ rather than the more accurate $\mathbf{int}((a,m),x)$.

Then, we may specify the Gödel number of the type **int** as,

$$\texttt{rep}(n, \textbf{int}) \;=\; n =_{tree} \texttt{tree}_0(1) \qquad \overset{1}{\bullet}$$

A similar attempt at specifying the Gödel number of the type $\tau_1 \cup \tau_2$ fails: the two occurrences of `rep` in the body of the following definition indicate that the `rep` relation is recursive.

$$\begin{array}{l}\texttt{rep}(n, \tau_1 \cup \tau_2)) \;= \\ \quad \exists n_1, n_2.\ \texttt{rep}(n_1, \tau_1) \wedge \texttt{rep}(n_2, \tau_2) \\ \qquad\quad \wedge\, n =_{tree} \texttt{tree}_2(2, n_1, n_2)\end{array} \qquad \begin{array}{c}2 \\ \diagup\ \diagdown \\ n_1 \quad n_2\end{array}$$

An inductive specification of `rep` seems possible, though — to fix the above problem we simply require that the types τ_1 and τ_2 both be "smaller" than the type $\tau_1 \cup \tau_2$. Consider the following inductive definition of `rep` which specifies the Gödel numbers of the types **int**, $\tau_1 \cup \tau_2$ and **ref** τ_1. (Note that we use ρ for `rep` variables.)

$$\begin{array}{l}\texttt{rep}(n, \tau) \;= \\ \quad \forall \rho.\ \rho(\texttt{tree}_0(1), \textbf{int}) \\ \qquad \wedge (\forall n_1, n_2.\ \rho(n_1, \tau_1) \wedge \rho(n_2, \tau_2) \\ \qquad\qquad \Rightarrow \rho(\texttt{tree}_2(2, n_1, n_2), \tau_1 \cup \tau_2)) \\ \qquad \wedge (\forall n_1.\ \rho(n_1, \tau_1) \;\Rightarrow\; \rho(\texttt{tree}_1(3, n_1), \textbf{ref}\ \tau_1)) \\ \quad \Rightarrow \rho(n, \tau)\end{array}$$

Definitions of the types **int**, $\tau_1 \cup \tau_2$ and **ref** τ_1 (which would, of course, precede the above definition of `rep`) are given below:

$$\begin{array}{lcl}\textbf{int}(a, m, x) & = & \texttt{true} \\ \tau_1 \cup \tau_2(a, m, x) & = & \tau_1(a, m, x) \vee \tau_2(a, m, x)\end{array}$$

We say $x :_{a,m}$ **ref** τ if location x is allocated and may only be updated with values of type τ, and if the value stored in memory at location x is of type τ: [7]

$$\textbf{ref}\ \tau(a, m, x) \;=\; \exists n.\ (x, n) \in a \wedge \texttt{rep}(n, \tau) \wedge \tau(a, m, m(x))$$

Unfortunately, when we consider the definitions of **ref** and `rep` simultaneously, we realize there is a problem: these definitions are mutually recursive.

3.3 3rd Flawed Model: The Gödel Number of `rep`

We can try to eliminate the mutual recursion by parametrizing the type constructor **ref** by `rep`. Then our definition of `rep` would have to specify the Gödel number of **ref**$(\texttt{rep}, \tau_1)$. A possible representation is $\texttt{tree}_2(3, n_\rho, n_1)$ where n_ρ is the Gödel number of `rep` and n_1 is the Gödel number of τ_1. It turns out, however, that if a Gödel number for `rep` exists, then (by a diagonalization argument) we can prove our logic inconsistent (i.e., $\exists n_\rho.\ \ \texttt{rep}(n_\rho, \texttt{rep}) \Rightarrow \texttt{false}$). Informally, if `rep` can represent both itself and the negation predicate ($\neg$), then it can represent "this sentence is false." The role of negation $\neg\tau$ is played [5] by the continuation constructor, **codeptr**(τ), defined in section 5.3.

Thus far it seems that our definition of mutable references is circular. Let us take a closer look at the judgment $l :_{a,m}$ **ref**$(\textbf{int})$:

$$\begin{array}{l}\textbf{ref}(\texttt{rep}, \textbf{int}))(a, m, l) \;\equiv \\ \quad \exists n.\ (l, n) \in a \wedge \texttt{rep}(n, \textbf{int}) \wedge \textbf{int}(a, m, m(l))\end{array}$$

[7] Recall that the allocset is a finite map from locations to Gödel numbers. Since a finite map can be modeled as a relation, we write $(x, n) \in a$ rather than $a(x) = n$.

Here `rep` is used solely to determine the Gödel number of **int** — this is a "smaller" type than **ref**$(\textbf{int})$. In fact, to determine the members of the set **ref** τ we only consider only those locations in the allocset whose permissible update types are "smaller." Rather than parametrize **ref** τ by `rep`, it would be sufficient to parametrize it by a version of `rep` that only specified the Gödel numbers of types "smaller" than **ref** τ.

3.4 Solution: A Hierarchy of Gödel Numberings

We use a stratified Gödel numbering relation $\texttt{rep}(i)(n, \tau)$ where the Gödel number n represents the type τ at level i. A property of this relation is that for all i $\texttt{rep}(i) \subset \texttt{rep}(i+1)$. Figure 2 illustrates the first few levels of the hierarchy for the type constructors **int**, $\cup$, and **ref**. Level 0 consists of the Gödel numberings of **int**, **int**$\cup$**int**, (**int**$\cup$**int**)$\cup$**int**, and so on. Let τ^0 denote a type that has a Gödel number at level 0. Then, level 1 consists of Gödel numberings of types τ^0, of types **ref** τ^0, and of all types in the closure (with respect to $\cup$) of all the level 1 types just mentioned.

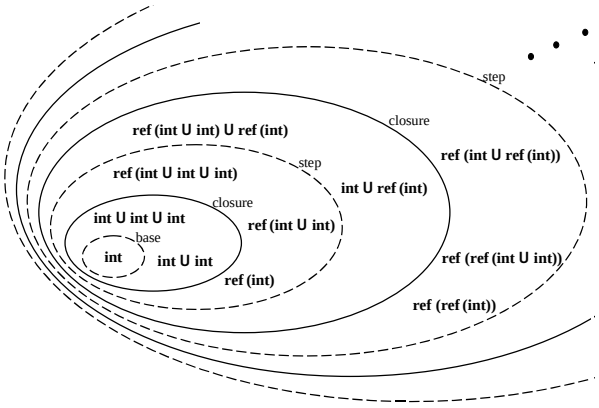


Figure 2: Hierarchy of Gödel Numberings

Figure 2 also shows how the `rep` relation may be inductively specified using the relations `base`, `closure` and `step`. Here, $\texttt{base}(n, \tau)$, which is a subset of $\texttt{rep}(0)$, specifies the Gödel numbers n of all primitive types τ; the relation $\texttt{closure}(\rho)(n, \tau)$ (where ρ is a subset of $\texttt{rep}(i)$) specifies $\texttt{rep}(i)$; the relation $\texttt{step}(i, \rho)(n, \tau)$ (where ρ is $\texttt{rep}(i)$) defines a subset of $\texttt{rep}(i+1)$; the `closure` of the latter, then, gives us $\texttt{rep}(i+1)(n, \tau)$. The formal definition of `rep` (for a small set of types) appears in figure 3, along with the relevant type definitions. We use the variable ρ for both `rep` and $\texttt{rep}(i)$.

Note that the type constructor **ref** is parametrized by $\texttt{rep}(i)$, as indicated by the formal parameter ρ. By creating a hierarchy of Gödel numbers, we have, in effect, created a hierarchy of mutable reference types: to define the set **ref** τ at level $i+1$ we need to know the members of all the sets **ref** τ' at levels 0 through i — for this we use $\texttt{rep}(i)$. By stratifying mutable references we have eliminated the circularity.

4. What is a Type?

Thus far we have described a type as a predicates on a state (a, m) and an integer x. But a predicate on (a, m, x) must have certain properties in order to be considered a type in our semantic model. To specify these properties, we need some definitions.

4.1 Valid States

$$
\begin{array}{lcl}
\textbf{int}(a,m,x) & = & \text{true} \\
\tau_1 \cup \tau_2(a,m,x) & = & \tau_1(a,m,x) \ \vee \ \tau_2(a,m,x) \\
\textbf{ref}(r)(\tau)(a,m,x) & = & \exists n.\ (x,n) \in a \ \wedge \ \rho(n,\tau) \ \wedge \ \tau(a,m,m(x)) \\
 & & \\
\texttt{base}(n,\tau) & = & n =_{tree} \texttt{tree}_0(1) \ \wedge \ \tau =_{type} \textbf{int} \\
\texttt{closure}(\rho)(n,\tau) & = & \forall \rho'.\ \rho \subset \rho' \\
 & & \quad \wedge \forall n_1,\tau_1,n_2,\tau_2.\ \rho'(n_1,\tau_1) \ \wedge \ \rho'(n_2,\tau_2) \Rightarrow \rho'(\texttt{tree}_2(2,n_1,n_2),\tau_1 \cup \tau_2) \\
 & & \quad \Rightarrow \rho'(n,\tau) \\
\texttt{step}(i,\rho)(n,\tau) & = & \rho(n,\tau) \ \vee \ \exists n_1,\tau_1.\ \rho(n_1,\tau_1) \ \wedge \ n =_{tree} \texttt{tree}_2(3,\texttt{tree}_0(i),n_1) \ \wedge \ \tau =_{type} \textbf{ref}(\rho,\tau_1) \\
\texttt{rep}(i)(n,\tau) & = & \forall \rho.\ \rho(0) = \texttt{closure}(\texttt{base}) \\
 & & \quad \wedge \forall j < i.\ \rho(j+1) = \texttt{closure}(\texttt{step}(j,\rho(j))) \\
 & & \quad \Rightarrow \rho(i)(n,\tau)
\end{array}
$$

Figure 3: Gödel Numbering of Types (int, $\cup$, ref)

The judgment $x :_{a,m} \tau$ says that x has type τ with respect to the state (a,m). Implicit in this assertion is the assumption that state (a,m) is well-formed or *valid*. We say that a state (a,m) is valid if it satisfies three conditions:

- The allocset a is a partial function, that is, each location in the allocset is mapped to only one Gödel number.
- The allocset a contains only legitimate Gödel numbers, which we define as,

  $$\texttt{godel}_\rho(n) \ = \ \exists \tau.\, \rho(n,\tau)$$

  The subscript ρ indicates that `godel` takes $\texttt{rep}(i)$ as an argument (where the level i is determined by the context, as we shall describe in section 5.3)).
- The type of the value stored at each allocated location in memory m matches its permissible update type in a; we say that x matches a Gödel number n with respect to a state (a,m) if,

  $$\texttt{match}_\rho(n,a,m,x) \ = \ \forall \tau.\, \rho(n,\tau) \Rightarrow \tau(a,m,x)$$

Definition 4 (Valid State)

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

A note on notation: we shall omit the subscript ρ and write $\texttt{validstate}(a,m)$ (and similarly for `godel` and `match`) to indicate that the missing parameter ρ should be instantiated with the default value $\overline{\rho}$, where,

$$\overline{\rho}(n,\tau) \ = \ \exists i.\ \texttt{rep}(i)(n,\tau)$$

As a consequence of our definition of `validstate` we have the following property. Informally, the type of the memory contents of an allocated address matches the type that the allocset says it should have:

Lemma 5 (Contents of ref Well-Typed)

$$\frac{(x,n) \in a \qquad \texttt{validstate}(a,m)}{\exists i,\tau.\ m(x) :_{a,m} \tau \ \wedge \texttt{rep}(i)(n,\tau)}$$

4.2 Valid State Extension

To formally describe the memory and allocset extensions permissible in our model we specify the extend-state relation $(a,m) \sqsubseteq_\rho (a',m')$ which says that state (a',m') is a valid extension of state (a,m) — or, alternatively, that (a,m) approximates (a',m'). State extensions must satisfy the following constraints:

- Our model does not permit deallocation of memory, so if $x \in \text{dom}(a)$, then we require that $x \in \text{dom}(a')$.
- The permissible update type of an allocated location cannot be altered across state extensions, so if $(x,n) \in a$ then $(x,n) \in a'$.
- The model requires that all memory updates be type-preserving — the last two conditions ensure that all allocated locations are "preserved" under state extension, so now we simply require that state (a',m') be a valid state.

Definition 6 (Extend State)
Valid state extension ($\sqsubseteq_\rho$) is specified as,

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

Again, we omit the subscript ρ to indicate $\rho = \overline{\rho}$.

Lemma 7 ($\sqsubseteq_\rho$ Transitive)
The extend-state relation ($\sqsubseteq_\rho$) is transitive.

A state extension $(a,m) \sqsubseteq_\rho (a',m')$ requires that both (a,m) and (a',m') be valid states. 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 state $(a,m) \sqsubseteq_\rho (a',m')$, the *Allocated Memory Well-Typed* property of a valid state forces us to initialize a new memory location (with a value of the appropriate type) *before* we add it to the allocset:

Lemma 8 (Initialization Before Allocation)

$$(a,m) \sqsubseteq_\rho (a,m') \Leftrightarrow (a,m) \sqsubseteq_\rho (a,m') \sqsubseteq_\rho (a',m')$$

4.3 Desirable Properties of Types

In section 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 is closed under valid extension of the memory. (Recall that $x :_{a,m} \tau$ is just syntactic sugar for $\tau(a,m,x)$.)

$$\texttt{update-inv}_\rho(\tau) = \forall x,a,m,m'.\ (a,m) \sqsubseteq_\rho (a,m') \wedge x :_{a,m} \tau \Rightarrow x :_{a,m'} \tau$$

Notice that since $(a,m) \sqsubseteq_\rho (a,m')$ allows updates of both allocated and unallocated locations, the above property incorporates the notion of *Initialization Invariance* described in section 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:

$$\texttt{alloc-inv}_\rho(\tau) = \forall x,m,a,a'.\ (a,m) \sqsubseteq_\rho (a',m) \wedge x :_{a,m} \tau \Rightarrow x :_{a',m} \tau$$

We say that a predicate τ (that takes an allocset, a memory, and an integer) is a `type` (written τ `type`) if it has both the $\texttt{update-inv}_\rho$ and $\texttt{alloc-inv}_\rho$ properties:

Definition 9

$$\texttt{type}(\tau) = \forall \rho.\texttt{update-inv}_\rho(\tau) \wedge \texttt{alloc-inv}_\rho(\tau)$$

From the definitions of $\texttt{update-inv}_\rho$ and $\texttt{alloc-inv}_\rho$, and the *Initialization Before Allocation* property of state extension ($\sqsubseteq_\rho$) we can show that τ is a `type` if and only if it is closed under state extension:

Lemma 10 (Type)

$$\texttt{type}(\tau) \equiv \forall \rho,a,m,a',m'.\ (a,m) \sqsubseteq_\rho (a',m') \Rightarrow \tau(a,m,x) \Rightarrow \tau(a',m',x)$$

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

Representability. Suppose that we wish to allocate a new reference cell that contains a value of type τ. We first write the value into unallocated memory, say at address l. Then, we extend the allocset with the pair (l,n) where n represents τ at some level i in the Gödel numbering hierarchy. Clearly this last step requires the existence of such an n and i; that is, to construct a value of type τ we require that τ be representable:

Definition 11 (Representable)

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

5. 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. We first explain some of the more involved type definitions, then specify a Gödel numbering relation for the types in our system, and finally present two sets of theorems: we show for each type in our system that type judgments are preserved under state extension (i.e., $\texttt{type}(\tau)$) and that the type is representable at some level in the Gödel numbering hierarchy (i.e., $\texttt{repable}(\tau)$).

$$\begin{array}{lcl}
\top(a,m,x) & = & \texttt{true} \\
\bot(a,m,x) & = & \texttt{false} \\
\textbf{int}(a,m,x) & = & \texttt{true} \\
\textbf{const}(n)(a,m,x) & = & x = n \\
\textbf{char}(a,m,x) & = & 0 \le x < 256 \\
\textbf{boxed}(a,m,x) & = & x \ge 256 \\
\textbf{offset}(i,\tau)(a,m,x) & = & \tau(a,m,x+i) \\
(\tau_1 \cup \tau_2)(a,m,x) & = & \tau_1(a,m,x) \vee \tau_2(a,m,x) \\
(\tau_1 \cap \tau_2)(a,m,x) & = & \tau_1(a,m,x) \wedge \tau_2(a,m,x) \\
\textbf{rec}(F)(a,m,x) & = & \forall \tau.\texttt{type}(\tau) \Rightarrow F(\tau) \subset \tau \Rightarrow \tau(a,m,x) \\
(\exists F)(a,m,x) & = & \exists \tau.\texttt{type}(\tau) \wedge (F\tau)(a,m,x) \\
(\forall F)(a,m,x) & = & \forall \tau.\texttt{type}(\tau) \Rightarrow (F\tau)(a,m,x) \\
\textbf{box}(\tau)(a,m,x) & = & \exists n.(x,n) \in a \wedge \texttt{base}(n,\textbf{const}(m(x))) \wedge \tau(a,m,m(x)) \\
\textbf{ref}(\rho)(\tau)(a,m,x) & = & \exists n.(x,n) \in a \wedge \rho(n,\mathbf{K}\tau) \wedge \tau(a,m,m(x)) \\
\textbf{codeptr}(\rho)(i,\tau)(a,m,x) & = & \forall r',m',a',\rho'.r'(\text{PC}) = x \wedge \rho \subset \rho' \wedge (a,m) \sqsubseteq_{\rho'} (a',m') \wedge a' = \texttt{alloc}(r',m') \wedge \tau(a',m',r'(i)) \Rightarrow \texttt{safe-state}(r',m')
\end{array}$$

Figure 4: Type Definitions

5.1 Recursive Types (rec)

To define recursive types, we introduce a subtyping relation defined as logical implication:

$$\tau_1 \subset \tau_2 = \forall a,m,x.\tau_1(a,m,x) \Rightarrow \tau_2(a,m,x)$$

A predicate $\textbf{rec}(F)$, where F is a function from types to types, is a `type` if the least fixed point of the argument function F is $\textbf{rec}(F)$. This property holds if the type function F preserves the validity of a type and if it is monotone. We define these properties, as well as antimonotonicity, as follows:

$$\begin{array}{lcl}
\texttt{preserve-type}(F) & = & \forall \tau.\texttt{type}(\tau) \Rightarrow \texttt{type}(F(\tau)) \\
\texttt{monotone}(F) & = & \forall \tau_1,\tau_2.\tau_1 \subset \tau_2 \Rightarrow F(\tau_1) \subset F(\tau_2) \\
\texttt{antimonotone}(F) & = & \forall \tau_1,\tau_2.\tau_1 \subset \tau_2 \Rightarrow F(\tau_2) \subset F(\tau_1)
\end{array}$$

We can prove that the composition of two antimonotone functions is monotone, while the composition of a antimonotone with a monotone, and vice versa, is antimonotone.

Note that in our current model, we cannot prove $\texttt{type}(\textbf{rec}(F))$ when F is antimonotone, i.e., we cannot handle contravariant recursive types; though we have built a more complicated model that does handle the latter (see section 7.1) we have chosen to explain our approach in a simpler setting.

5.2 Immutable References (box)

The type **box** of immutable references is defined in figure 4. Any value may be stored in an immutable location, as long as it is numerically equal to the value that's already there.

5.3 First-Order Continuations (codeptr)

A value of type $\textbf{codeptr}(i,\tau)$ is a first-order continuation, that is, a machine-code address to which control may be passed, provided that its preconditions are met. Consider the judgment $x :_{a,m}$

codeptr$(\rho)(i,\tau)$. This says that if we jump to address x in some future machine state (r',m') (i.e., if $r'((pc)) = x$), then (r',m') is a safe state if the following conditions hold:

- The argument (i.e, the value stored in register $\mathbf{r}_i$) must be of the right type; that is, $r'(i) :_{a',m'} \tau$, where a' is computed as usual from the machine state (r',m') as follows: $a' = \texttt{alloc}(r',m')$.
- 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. Then, since the program is preserved under state extension, our precondition is simply: $(a,m) \sqsubseteq_{\rho'} (a',m')$, where ρ' satisfies the condition we describe next.
- $\rho \subset \rho'$. Informally, if $\rho = \texttt{rep}(i)$, where ρ is used to ensure $\texttt{validstate}_\rho(a,m)$, and if $\rho' = \texttt{rep}(j)$, where ρ' used to ensure $\texttt{validstate}_{\rho'}(a',m')$ (see previous condition), then we require that $j \geq i$. In other words, 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 to ensure that state (a',m') is valid.

5.4 Gödel Numbers of Type Functions

Our model requires that we specify the Gödel number of each of the types in figure 4. But to specify the Gödel numbers of **rec**(F), $\exists F$, or $\forall F$ (where F is function from types to types), we have to first specify the Gödel numbers of type functions F. Rather than extend the `rep` relation so that it specifies the Gödel numbers of type functions as well as types, we observe that if we represent every type as a type function that simply ignores its argument, then we would only require Gödel numbers of type functions. We use $\mathbf{K}\tau$ to denote a type function that ignores its argument and returns the type τ (i.e., $\mathbf{K} = \lambda y.\lambda x.y$). The Gödel numbering relation for a representative set of type functions appears in figure 5 — we do not present the full `rep` relation due to space limitations.

Note that the relation $\texttt{base}(n,F)$ (figure 5) specifies the Gödel numbers of $\mathbf{K}\,\tau$ for primitive types τ, as well as, the Gödel number of the identity type function $\lambda\tau.\tau$. All type constructors, with the exception of **ref** and **codeptr**, are used to compute the `closure` of each $\texttt{rep}(i)$. For example, if F is representable at level i, then $\lambda\tau.\mathbf{ref}(F(\tau))$ and $\lambda\tau.\mathbf{codeptr}(i,F(\tau))$ are representable at level $i+1$, as are $\mathbf{K}\,\mathbf{rec}(\lambda\tau.\mathbf{ref}(F(\tau)))$ and $\lambda\tau.(\mathbf{int}\cup\mathbf{codeptr}(i,F(\tau)))$.

5.5 Theorems: `type` and `repable`

Theorem 12 (τ `type`)
Each of our types is a `type`*, e.g.:*

a-f. $\texttt{type}(\tau)$, *where* $\tau ::= \top|\bot|\mathbf{int}|\mathbf{const}(n)|\mathbf{char}|\mathbf{boxed}$

g. $\texttt{type}(\tau) \Rightarrow \texttt{type}(\mathbf{offset}(i,\tau))$.

h. $\texttt{type}(\tau_1) \wedge \texttt{type}(\tau_2) \Rightarrow \texttt{type}(\tau_1 \cup \tau_2)$.

i. $\texttt{type}(\tau_1) \wedge \texttt{type}(\tau_2) \Rightarrow \texttt{type}(\tau_1 \cap \tau_2)$.

j. $\texttt{preserve-type}(F) \wedge \texttt{monotone}(F) \Rightarrow \texttt{type}(\mathbf{rec}(F))$.

k-l. $\texttt{preserve-type}(F) \Rightarrow \texttt{type}(\tau)$, *where* $\tau ::= \exists F|\forall F$

m. $\texttt{type}(\tau) \Rightarrow \texttt{type}(\mathbf{box}(\tau))$

n. $\texttt{type}(\tau) \Rightarrow \texttt{type}(\mathbf{ref}(\rho)(\tau))$

o. $\texttt{type}(\tau) \Rightarrow \texttt{type}(\mathbf{codeptr}(\rho)(i,\tau))$.

Proof: Immediate from the definitions of the type operators. □

Appel and Felty [2] show how to define record types using the types **ref**, **offset** and $\cap$ as building blocks. To define function closures — where a function of type $\alpha \to \beta$ takes an argument of type α, a continuation of type β and an environment — we require record types, a codeptr type, and the existential type. Since we have shown $\texttt{type}(\tau)$ for all the types in our system, any type constructible from the types shown in figure 4 (except arbitrarily nested recursive and quantified types — see section 7.2) can easily be shown to be a `type`.

Theorem 13 (Representable Types are Valid)
$\texttt{rep}(n,\tau) \Rightarrow \texttt{type}(\tau)$

Proof: By a nested induction argument. The outer induction proceeds on the representation level, and it aims to show that validity of representable types is preserved by `step`. The inner induction is on the structure of the type τ (or, equivalently, on that of the corresponding Gödel number n), and aims to show that, if all types represented by a representation function ρ are valid, so are those represented by $\texttt{closure}(\rho)$. □

Lemma 14 (`rep` Upward Closed)
$\texttt{rep}(i,n,\tau) \Rightarrow \texttt{rep}(i+1,n,\tau)$

Proof: It follows from the definition of `rep` and the following two basic facts:

$$\forall\rho.\ \rho \subset \texttt{closure}(\rho)$$
$$\forall\rho, j.\ \rho \subset \texttt{step}(j,\rho)$$

□

Theorem 15 ($\texttt{repable}(\tau)$)
Each of our types is representable: $\exists i,n.\texttt{rep}(i,n,\tau)$.

Proof: This proof consists of computing, by induction on the structure of τ, the minimum level i at which τ is representable. □

5.6 Invariance Lemmas

Lemma 16 (Initialization Invariance)

$$\frac{x :_{a,m} \tau \qquad y \notin a \qquad m' = m[y:=z] \qquad \texttt{type}(\tau)}{x :_{a,m'} \tau}$$

Lemma 17 (Allocation Invariance)

$$\frac{x :_{a,m} \tau \qquad (a,m) \sqsubseteq (a',m) \qquad \texttt{type}(\tau)}{x :_{a',m} \tau}$$

Lemma 18 (Update Invariance)

$$\frac{x :_{a,m} \tau \quad y :_{a,m} \mathbf{ref}\,\tau' \quad z :_{a,m} \tau' \quad m' = m[y:=z] \quad \texttt{type}(\tau)}{x :_{a,m'} \tau}$$

Proof: Lemmata 16-18 are proved by a nested induction argument similar to the one employed in the proof of theorem 13. □

$$
\begin{array}{lcl}
\texttt{base}(n,F) & = & n =_{tree} \texttt{tree}_0(10) \wedge F =_{tyfn} \mathbf{K}\,\mathbf{int} \\
 & & \vee\ n =_{tree} \texttt{tree}_1(11, \texttt{tree}_0(c)) \wedge F =_{tyfn} \mathbf{K}(\mathbf{const}(c)) \\
 & & \vee\ n =_{tree} \texttt{tree}_0(12) \wedge F =_{tyfn} \lambda\tau.\tau \\
\texttt{closure}(r,n,F) & = & \forall \rho'.\, \rho \subset \rho' \\
 & & \quad \wedge \forall n_1,F_1,n_2,F_2.\, \rho'(n_1,F_1) \wedge \rho'(n_2,F_2) \Rightarrow \rho'(\texttt{tree}_2(20,n_1,n_2), \lambda\tau.F_1(\tau) \cup F_2(\tau)) \\
 & & \quad \wedge \forall n_1,F_1.\, \texttt{monotone}(F_1) \wedge \texttt{preserve-type}(F_1) \wedge \rho'(n_1,F_1) \\
 & & \quad\quad \Rightarrow \rho'(\texttt{tree}_1(21,n_1), \mathbf{K}\,\mathbf{rec}(F_1)) \\
 & & \quad \Rightarrow \rho'(n,F) \\
\texttt{step}(i,\rho,n,F) & = & \rho(n,F) \\
 & & \vee \exists n_1,F_1.\, \rho(n_1,F_1) \wedge n =_{tree} \texttt{tree}_2(30, \texttt{tree}_0(i), n_1) \wedge F =_{tyfn} \lambda\tau.\, \mathbf{ref}(\rho, F_1(\tau)) \\
 & & \vee \exists n_1,F_1.\, \rho(n_1,F_1) \wedge n =_{tree} \texttt{tree}_2(31, \texttt{tree}_0(i), \texttt{tree}_0(j), n_1) \wedge F =_{tyfn} \lambda\tau.\, \mathbf{codeptr}(\rho, j, F_1(\tau)) \\
\texttt{rep}(i)(n,F) & = & \forall \rho.\, \rho(0) = \texttt{closure}(\texttt{base}) \\
 & & \quad \wedge \forall j < i.\, \rho(j+1) = \texttt{closure}(\texttt{step}(j, \rho(j))) \\
 & & \Rightarrow \rho(i)(n,F)
\end{array}
$$

Figure 5: Gödel Numbering of Type Functions (int, const, ∪, rec, ref, codeptr)

5.7 Introduction & Elimination Rules

We prove introduction and elimination lemmas for each of the types shown in figure 4. Due to space limitations, we only show the lemmas for **ref** here.

Lemma 19 (ref_i and ref_e)

$$\frac{m(x) :_{a,m} \tau \qquad x \notin a}{\exists a'.\,(a,m) \sqsubseteq (a',m) \ \wedge\ x :_{a',m} \mathbf{ref}\,\tau}$$

$$\frac{x :_{a,m} \mathbf{ref}\,\tau}{m(x) :_{a,m} \mathbf{ref}\,\tau}$$

6. An Example

We present a program fragment that constructs a reference cell of type τ_2 in memory (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. We show how our model of general references allows us to prove $\texttt{preservation}(l)$ for each program point l in the fragment.

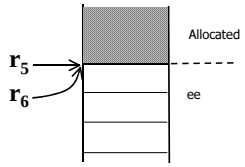


Figure 6: Example

Consider the situation in figure 6 as the starting point of the following program fragment. For this example, assume that register 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.[8]

[8] Appel and Felty [2] use a register (say r_6)to point to an allocation boundary such that all addresses less than r_6 are allocated and all addresses beyond that point are unallocated. Their single argument `alloc` predicate is defined as $\texttt{alloc}(r,m) = \lambda x.\, x < r(6)$.

$\{I_{111} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(4) :_{a,m} \tau_2$
$\qquad\qquad \wedge r(5) \notin \texttt{dom}(a) \wedge r(5) = r(6)\}$
$\mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4}$
$\{I_{112} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge m(r(5)) :_{a,m} \tau_2\}$
$\mathbf{r_6} \leftarrow \mathbf{r_6} + \mathbf{1}$
$\{I_{113} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(5) :_{a,m} \mathbf{ref}\,\tau_2\}$
$\mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4}$
$\{I_{114} = \lambda r,m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(5) :_{a,m} \mathbf{ref}\,\tau_2\}$

Proof: $\{I_{111}\}\mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4}\{I_{112}\}$
By the semantics of the store instruction and the step relation, for all $x \neq \text{PC}$, $r'(x) = r(x)$ and $m'(r(5)) = r(4)$.

- We have $r'(1) :_{a,m'} \tau_1$ using lemma 16. Since a store instruction does not alter the allocset, $a = a'$. Then $r'(1) :_{a',m'} \tau_1$ follows by congruence.
- $r'(4) :_{a',m'} \tau_2$ follows using the same argument as for $r'(1) :_{a',m'} \tau_1$ above. Then $m'(r'(5)) :_{a',m'} \tau_2$ follows by congruence.

□

Proof: $\{I_{112}\}\mathbf{r_6} \leftarrow \mathbf{r_6} + \mathbf{1}\{I_{113}\}$
Pick $a' = \lambda x,n.(x,n) \vee (x = 303 \wedge \exists i.\, \texttt{rep}(i)(n, \mathbf{K}\tau_2))$. It is easy to show that a' satisfies $(a,m) \sqsubseteq (a',m')$.

- Since the add instruction does not alter memory, $m' = m$. Then we have $r'(1) :_{a',m'} \tau_1$ using lemma 17 and congruence.
- Since register r_5 is unchanged, we have $r'(5) = 303$, so it would suffice to show that $303 :_{a',m'} \mathbf{ref}\,\tau_2$. (At this point we can use the **ref_i** lemma, but we proceed without it to clarify the details.) By theorem 15, we have $\exists i.\texttt{rep}(i)(n,\tau_2)$; then we can show that $(303,n) \in a'$. To prove $m(303) :_{a',m'} \tau_2$ we use the premise $m(r(5)) :_{a,m} \tau_2$ along with congruence and lemma 17.

□

Proof: $\{I_{113}\}\mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4}\{I_{114}\}$
This proof is analogous to the proof for the first instruction of the program fragment, except that lemma 18 replaces all uses of lemma 16 in that proof. □

7. Extensions

We describe some of the other extensions to the Appel-Felty model [2] and show that our model is compatible with these.

7.1 The Indexed Model

Consider the type $\mathbf{rec}(\lambda\alpha.\,\alpha \to \mathbf{int})$. This is a contravariant recursive type. In these types, occurrences of the type being defined appear *negatively*, that is, they appear to the left of an odd number of function arrows in an ML declaration. Using the notation of section 5.1, a type $\mathbf{rec}(F)$ is contravariant whenever F is `antimonotone`.

The indexed model of types described by Appel and McAllester [3] allows reasoning about contravariant recursive types, but in an immutable setting. In that model, types are predicates on (k, a, m, x) where k is an approximation index, and (a, m) and x are a state and a root-pointer, respectively. The judgment $x :_{k,a,m} \tau$ means, "x approximately has type τ, and any program that runs for fewer than k instructions can't tell the difference." Then, a program that executes j instructions where $j \leq k$ also *believes* that x has type τ; that is, $x :_{k,a,m} \tau \;\Rightarrow\; \forall j \leq k.\; x :_{j,a,m} \tau$.
The indices k allow the construction of a well-founded recursion, even when modeling contravariant recursive types.

It turns out that the approximation indices used in the indexed model can coexist quite naturally with the hierarchy of Gödel numberings that we have described. Informally, this is due to the nature of **ref**: suppose we have $x :_{k,a,m} (\mathbf{ref}\ \tau)^i$ — we have used the superscript i to denote that the Gödel number of **ref** τ is defined at `rep` level i. Since x is a pointer to a value $m(x)$ of type τ, and since it takes one execution step (i.e., one instruction) to dereference a pointer, we may conclude that any program that runs for fewer than $k-1$ instructions believes that $m(x)$ has type τ — this follows from the definition of **ref** in the indexed model. Then, we have $m(x) :_{k-1,a,m} \tau^{i-1}$, i.e, both the approximation index and the `rep` level are decremented by the definition of **ref**.

It is important to note that the k and i are different: k deals with the number of times a recursive type is unrolled, while i simply counts the number of nested occurrences of **ref** in a type expression

We have built a semantic model with both approximation indices k and a hierarchy of Gödel numberings, with the theorems of section 5 suitably modified for the indexed setting; for each of these theorems, we have machine-checked proofs in this model. Hence, we have a semantic model of general references that can handle both covariant and contravariant recursive types.

7.2 Typed Machine Language

A limitation of the semantics we described in section 5 is that it does not allow us to represent arbitrarily nested recursive and quantified types. (This limitation also applies to the model in section 7.1 for which we have a machine-checked proof.) For instance, we cannot represent the following type in our model:

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

Typed Machine Language, described by Swadi and Appel, [23] accommodates arbitrarily nested recursive and quantified types and it does so using DeBruijn indices. Our approach is compatible with the latter: we simply need to define a Gödel numbering relation that represents type expressions with free DeBruijn variables rather than type functions as we have shown. Though this requires a considerable proof-implementation effort, conceptually the task is not too complicated.

8. 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.

To do machine-checked proofs, one must first choose a logic and a logical framework in which to manipulate the logic. 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 [7] implemented in the Twelf logical framework [18]. Our proof "implementation" consists of about 16,500 lines of Twelf code, using the encoding of higher-order logic described by Appel and Felty [2]. The implementation of the Appel-McAllester model consisted of 8,200 lines of proof, while that of the Appel-Felty model consisted of 5,400 lines.

9. Related Work

A common feature of a number of models of mutable state (also called *mutable store* in the literature) is the following: they specify *how* the state is allowed to vary over time. Models for *Idealized Algol* developed by Reynolds and Oles [20, 15, 16, 17] make use of functor categories; functors are important because they capture the fact that the size of the store, as well as its contents, may change over time. To specify how the state is allowed to change at any point in the program, they use functor categories indexed by *possible worlds* or *store shapes*. We note, however, that these models do not handle general references. Stark [22] (building on work done with Pitts on possible world models of the nu-calculus [19]) describes a denotational semantics for *Reduced ML* that includes integer references.

Recently, Levy [10] has described a possible-world 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 worlds w, like our allocset a, can only increase (written $w \leq w'$ where w is the earlier world and w' is the later world). 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 `type` in our model. Moreover, the definition (denotation to be more precise) of the type **ref** τ in his model is different: $[\![\mathbf{ref}\ \tau]\!]w = \$w_\tau$ (where $\$w_\tau$ denotes "the set of cells of type τ in w"). Notice that $[\![\mathbf{ref}\ \tau]\!]$ is defined in terms of the syntax τ rather than the semantics $[\![\tau]\!]$; that is, this semantics is not compositional. Levy is faced with the same kind of circularity that we described in section 3. He solves it by observing that recursive equations on domains have solutions. We solve it by showing that our hierarchy of `rep` relations has a limit. This hints that Gödel numbering might be a way to model denotational semantics.

Besides Levy's, the only other model of general references that we know of is a game semantics described by Abramsky, Honda and McCusker [1]. The model of references is provided by certain categories of games: a strategy for an arena is a rule telling Player what move to make in a given position — *thread-independent* strategies specify how the state is allowed to vary over time. In this model, reference types are viewed as a product of a "read method" and a "write method" in the style of Reynolds [20]. This representation of references is quite different from that in location-based models such as ours.

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 [9] (which is closely related to the work of Burstall and Reynolds [4, 21]). 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 [14] 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 following rule:

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

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, for example, $x :_{a,m} \tau_1 \wedge y :_{a,m} \tau_2$ and $\mathit{ModifiesOnly}(C)$ is the set of registers that are updated by the command C:

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

The use of $\wedge$ instead of $*$ has important consequences. Consider the situation in figure 1. Now suppose we execute the following program fragment which updates the cell pointed at by register r_5. The following statement can be proved in our framework, but not in the theirs; there the heap cannot be split into two parts, such that one part satisfies $r(1) :_{a,m} \tau_1$ and the other satisfies $r(5) :_{a,m} \mathbf{ref}\ \tau$.

$$\begin{array}{l}\{I_{111} = \lambda r, m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(5) :_{a,m} \mathbf{ref}\ \tau_2 \\ \qquad\qquad \wedge\, r(4) :_{a,m} \tau_2\} \\ \mathbf{m}(\mathbf{r_5}) \leftarrow \mathbf{r_4} \\ \{I_{112} = \lambda r, m.\, \exists a.\ r(1) :_{a,m} \tau_1 \wedge r(5) :_{a,m} \mathbf{ref}\ \tau_2\}\end{array}$$

Their framework is useful where aliasing is not expected to occur because their predicates R are stronger than just typing judgments.

Melham [11] automated the definition and proof of recursive datatypes (without function types) in higher-order logic; Gunter [6] extended this to covariant function types in the recursion. Our Gödel numbering can be seen as an extension of this work to all functions representable by expressions. However, their systems generate the `rep` relation and proofs automatically; we have done it by hand but intend to automate the process in future work.

10. Future Work

Appel and McAllester's indexed model [3] has both a simple, non-extensional 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.

11. References

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

[3] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. Technical Report TR-629-00, Princeton University, October 2000.

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

[5] Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, New York, 1990. ACM Press.

[6] Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In Jeffery Joyce and Carl Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 141–154. Springer-Verlag, February 1994.

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

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

[9] Samin Ishtiaq and Peter 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.

[10] Paul Blain Levy. *Call-by-Push-Value*. Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001.

[11] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386, New York, 1989. Springer.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):527–568, May 1999.

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

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

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

[16] Frank Joseph Oles. Type algebras, functor categories, and block structure. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, Cambridge, England, 1985.

[17] Frank Joseph Oles. Functor categories and store shapes. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 2*, pages 3–12. Birkhäuser, Boston, Massachusetts, 1997.

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

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

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

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

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

[23] Kedar N. Swadi and Andrew W. Appel. Typed machine language and its semantics. submitted for publication, 2001.