A Very Modal Model of a Modern, Major, General Type System

Andrew W. Appel *
Princeton University
INRIA Rocquencourt
appel@princeton.edu

Paul-André Melliès
CNRS
Université Paris 7
mellies@pps.jussieu.fr

Christopher D. Richards*
Princeton University
richards@cs.princeton.edu

Jérôme Vouillon
CNRS
Université Paris 7
vouillon@pps.jussieu.fr

Abstract

We present a model of recursive and impredicatively quantified types with mutable references. We interpret in this model all of the type constructors needed for typed intermediate languages and typed assembly languages used for object-oriented and functional languages. We establish in this purely semantic fashion a soundness proof of the typing systems underlying these TILs and TALs—ensuring that every well-typed program is safe. The technique is generic, and applies to any small-step semantics including λ-calculus, labeled transition systems, and von Neumann machines. It is also simple, and reduces mainly to defining a Kripke semantics of the Gödel-Löb logic of provability. We have mechanically verified in Coq the soundness of our type system as applied to a von Neumann machine.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Modal logic

General Terms Languages, Theory

Keywords Recursive types, impredicative polymorphism, mutable references, Kripke models

1. Introduction

We wish to compile languages such as ML and Java into typed intermediate languages and typed assembly languages. These TILs and TALs are particularly difficult to design, because in order to describe the program transformations applied in the course of compilation, they require a very rich and expressive type system, including

- intersection and union types;
- types for address arithmetic;
- types for mutable references, since the source languages have mutable references (or object fields);
- recursive types, in order to describe data structures loaded in memory such as lists and trees;
- type quantification, which is used to describe object and function-closure representations.

Not only that: type quantification should be *impredicative*, because the types which may be abstracted (i.e., put in private fields of objects or in function-closure environments) can be themselves object or function-closure (i.e., quantified) types. In particular, it is not possible to stratify the type system into levels k such that a quantification $\forall_k \alpha.\tau$ is always instantiated at level $< k$.

Putting all these type ingredients together in a low-level language is an intricate exercise. A formal proof of *soundness*—any well-typed program does not go wrong—is thus recommended for any type system for such TILs and TALs.

Contributions. In this paper we will establish the soundness of a type system including all the ingredients mentioned earlier, and more. Our semantics of impredicative references is an improvement over previous work because it removes a cumbersome restriction from the typing rules for quantifiers (see Section 13.1). Our approach is also novel in that we have decomposed the semantic types in such a way that the principles of *safety*, *induction over the future*, *type preservation*, and so on are each identifiable as a specific operator of the type system. Standard concepts such as (function) code pointers, subtyping, contractiveness, mutable and immutable references are constructed from these new primitives.

Overview. Although our type system could be proved sound syntactically—using progress and preservation—we will not use progress and preservation. Hence we do not need to formally list all our type constructors and typing rules—because we never do induction over them—but the

* Supported in part by NSF grants CCF-0540914 and CNS-0627650.

To appear in *POPL'07*, January 17–19, 2007, Nice, France.

reader may examine Tables 1 and 2 and Figures 4–7. We will proceed in a purely semantic fashion, for two reasons at least. First, we find intuition easier to come by in this way: once types are interpreted in a relevant and felicitous way, it becomes much easier to combine or decompose them than if they remained ciphers rendered meaningful only by the typing rules.

Second, we enjoy the modularity and genericity of the semantic methods. We demonstrate this in the paper, with a nicely modularized proof of soundness which is applicable in principle to any calculus with a small-step semantics. We discuss here von Neumann machines and the call-by-value λ-calculus with references; our methods should also work well for labeled transition systems used in the intermediate representations of compilers.

To establish our soundness result, we construct a model of recursive and impredicatively quantified types with mutable references. We start from the idea of *approximation* which pervades all the semantic research in the area. If we type-check $v : \tau$ in order to guarantee safety of just the next k computation steps, we need only a k-approximation of the typing-judgment $v : \tau$. We express this idea here using a Kripke semantics whose possible-worlds accessibility relation R is well-founded: every path from a world w into the future must terminate. In this work, worlds characterize abstract computation states, giving an upper bound on the number of future computations steps and constraining the contents of memory. (We define the set of worlds precisely in Section 6.) We write

$$w \Vdash v : \tau$$

to mean that the value v has type τ in world w. This leads us to investigate a modal logic adapted to this Kripke semantics (Section 2). A judgment $\sigma \vdash \tau$ in this logic means that every value v of type σ at any world w has also type τ at that world w. Then (Section 4) a modal operator $\triangleright$ (pronounced "later") quantifies over all worlds (all times) *strictly* in the future. The strictness of $\triangleright$ turns out to be a key technical insight, if we compare it to previous work [Ahm04] which used instead a construction analogous to the modal operator necessity $\Box$ quantifying over worlds *now and* in the future. Indeed, the combination of a well-founded R with a strict modal operator $\triangleright$ provides a clean induction principle to the logic, called the *Löb* rule,

$$\frac{\triangleright\tau \vdash \tau}{\vdash \tau},$$

where $\vdash \tau$ can be read as asserting the safety of the program described by τ. (Equation 16 gives a precise definition.)

The resulting Gödel-Löb logic is ideal for computations that can't go on forever, and where information gets less and less precise as the computation goes on. Suppose that we want to type-check a program written for a von Neumann machine with arbitrary control flow (Section 11). Let τ give the types of all the labels appearing in the program. We want to show that τ is an induction hypothesis for the safety of the program, at least as far as the end of the worlds. We first assume $\triangleright\tau$; that is, to a weaker, or "later" approximation, the type τ holds for any exit label to which we might jump in (at least) one step. Then we analyze each basic block in the program to show that the type τ (that is, the stronger approximation, without $\triangleright$) describes each entry label. From this we would like to conclude by induction that τ absolutely describes the program, with no hypothesis; this is precisely what the Löb rule enables.

Of course, some computations *should* go on forever. But the proof scheme above works for any *finite prefix* of the computation—so the program is safe for an arbitrary number of steps. We obtain in this way (Section 12) a proof of soundness of the type system for a von Neumann machine—of which we have machine-checked proofs[1] in Coq, using a straightforward shallow embedding of our model.

Related work. Our model unifies and generalizes several previous threads of research: the indexed model [AM01, AAV02, Ahm04], the orthogonality model [MV05], and the approximation modality [Nak01]. We will discuss some of these in Section 13.

2. Kripke semantics

Kripke models [Kri63] originated as models of modal logics; subsequently they have been shown useful for giving the semantics of type systems [MM91]. The main idea behind Kripke models is that truth (respectively, membership in a type) is not absolute but relative to some appropriate notion of world, or state, and that these states can be related precisely to each other.

In this section we will interpret worlds as characterizing abstract properties of the current state of computation. In particular, in a system with mutable references, each world contains a memory typing (Section 6); but the model can be more general than that, so we will start with an abstract characterization.

Models. A *model* is a triple $(W, R, \Vdash)$, where W is a nonempty set of worlds w; an accessibility relation $R \subset W \times W$ describes when it is possible to move from one world to another; and $\Vdash$ assigns truth values to typing judgments with respect to a given world, that is,

$$w \Vdash v : \tau. \tag{1}$$

Here $w \in W$ is a world, $v \in V$ is a *value* of our operational calculus, and $\tau \in \mathrm{Type}$ is a type.

We apply our Kripke models to small-step operational semantics with a memory or store indexed by locations Loc; but there are still many flavors of such semantics:

[1] Available at `http://www.pps.jussieu.fr/~vouillon/smot/`

Notation	Eqn.	Description
top	2	Top type
bot	3	Bottom type
$\tau \wedge \sigma$	4	Intersection type
$\tau \vee \sigma$	5	Untagged union type
$\tau \Rightarrow \sigma$	6	Implication type
$\forall x{:}A.\tau$	7	Universal type
$\exists x{:}A.\tau$	8	Existential type
$!\tau$	9	Configs where all values have τ
$?\tau$	10	Configs where some value has τ
$\mathsf{just}\, u$	13	Singleton type
$\mathsf{slot}(j, \tau)$	15	Von Neumann vector element type
$\triangleright \tau$	18	"Later" modality
$\mathsf{rec}\, F$	24	Recursive type
$l \mapsto \tau$	26	Configs where $\Psi(l) = \tau$
$\mathsf{plus}(i, j, k)$	57	Asserts $i + j = k$
$\mathsf{greater}(i, j)$	58	Asserts $i > j$
$\mathsf{safemem}(m)$	65	Type of configs safe for memory m
unit	70	Unit type

Figure 1. Primitive types, operators

Notation	Eqn.	Description
$\tau \Longleftrightarrow \sigma$	11	Type of configs in τ iff in σ
$\mathsf{teq}(\tau, \sigma)$	12	Asserts τ equal to σ
$u \circledcirc \tau$	14	Asserts u has τ in current world
$\Box \tau$	19	Necessity modality
$\mathsf{validmem}(m)$	27	Asserts m valid
$\mathsf{ref}\, \tau$	32	Mutable reference type
$\mathsf{offset}(n, \tau)$	59	Pointer arithmetic type
$\mathsf{pair}(\sigma, \tau)$	60	Pair type
$\mathsf{list}\, \tau$	61	List type
$\mathsf{iref}\, \tau$	62	Immutable reference type
safe	66	Type of safe configs
$\tau \to \sigma$	71	Function type
$\mathsf{codeptr}(\tau)$	72	Code pointer (continuation) type
Δ_p	75	Program code type
Γ_p	79	Program invariants type
$\mathsf{boundary}\, l$	88	Asserts memory $> l$ unallocated

Figure 2. Synthesized types, operators

- In λ-calculus with mutable references, a *state* is a pair (e, m) where e is an expression and m is a *store*, a mapping from Loc to V. States evolve by a small-step relation $(e, m) \rightarrowtail (e', m')$; if there is no such (e', m') then either e is a value v or (e, m) is stuck. A world w characterizes the store m.
- On a von Neumann machine, the state is a pair (v, m), where $v \in V$ is a vector of Loc representing pointers in registers, including the program counter. The memory m is a mapping from Loc to Loc containing program and data. The small-step relation is $(v, m) \rightarrowtail (v', m')$. A world w characterizes the contents of memory.
- In a labeled transition system such as is used in flow analysis, w characterizes the store and v contains both the local variables and the control-flow state.

Clearly, the same value v may or may not have type τ depending on the world w, that is, depending on the data structures in memory that v points to. Accordingly, we call a pair (w, v) a configuration (abbreviated "config"):

$$\mathrm{Config} = W \times V,$$

and define a type $\tau \in \mathrm{Type}$ as a set of configurations. Then,

$$(w, v) \in \tau \qquad \text{and} \qquad w \Vdash v : \tau$$

are two alternative notations expressing the same fact. We will show how our semantics connects the relation R between worlds and the relation $\rightarrowtail$ between states.

3. Primitive type constructors

Our approach to the modeling of a type system is to define a few primitive type constructors—a type is just any set of configurations—and then use these primitives to synthesize all the constructors needed for type-checking programs.

We can start with intersection types, union types, and their identities, as well as an implication type:

$$\mathsf{top} \stackrel{\text{prim}}{=} \{(w, v) \mid \mathrm{True}\} \tag{2}$$

$$\mathsf{bot} \stackrel{\text{prim}}{=} \{\} \tag{3}$$

$$\tau \wedge \sigma \stackrel{\text{prim}}{=} \tau \cap \sigma \tag{4}$$

$$\tau \vee \sigma \stackrel{\text{prim}}{=} \tau \cup \sigma \tag{5}$$

$$\sigma \Rightarrow \tau \stackrel{\text{prim}}{=} \{(w, v) \mid (w, v) \in \sigma \Rightarrow (w, v) \in \tau\} \tag{6}$$

We write $\mathsf{c}(\tau) \stackrel{\text{prim}}{=} \ldots\tau\ldots$ to indicate that the primitive type-constructor c is defined by the mathematical expression $\lambda\tau.\ \ldots\tau\ldots$ in the underlying logic. For our machine-checked proofs in Coq [B+98], the underlying logic is the Calculus of Inductive Constructions [PM93].

Quantification. Quantification over types is needed for polymorphism, data abstraction, and representation of closures. Besides, we are interested here in a λ-calculus or von Neumann machine equipped with a store (memory) of mutable references with address arithmetic. To reason about dataflow and addressability in such programs, our type system should also be able to quantify over locations $l \in \mathrm{Loc}$ and memories $m \in \mathrm{Mem}$. Our quantifiers are thus,

$$\forall x{:}A.\tau \stackrel{\text{prim}}{=} \bigcap_{a \in A} \tau[a/x] \tag{7}$$

$$\exists x{:}A.\tau \stackrel{\text{prim}}{=} \bigcup_{a \in A} \tau[a/x] \tag{8}$$

where A is either Type, Loc, or Mem.

Quantification over values in a world. Another kind of quantification is over values in the current[2] world:

$$!\tau \stackrel{\text{prim}}{=} \{(w,v) \mid \forall v'.\,(w,v') \in \tau\} \tag{9}$$

$$?\tau \stackrel{\text{prim}}{=} \{(w,v) \mid \exists v'.\,(w,v') \in \tau\} \tag{10}$$

The type $!\tau$ ("everywhere τ") means that every value in the current world has type τ; and $?\tau$ ("somewhere τ") means that some value in the current world has type τ.

One strength of our approach is that we can define just a few primitive type constructors, and from those we can synthesize many more constructors of interest. We will use the notation $\mathsf{c}(\tau) \stackrel{\text{syn}}{=} \ldots\tau\ldots$ to mean that the constructor c is synthesized from a combination $\ldots\tau\ldots$ of our type constructors, without dipping into the underlying logic.

$$\tau \Longleftrightarrow \sigma \stackrel{\text{syn}}{=} \tau \Rightarrow \sigma \wedge \sigma \Rightarrow \tau \tag{11}$$

$$\mathsf{teq}(\tau,\sigma) \stackrel{\text{syn}}{=} !(\tau \Longleftrightarrow \sigma) \tag{12}$$

The judgment $w \Vdash v : \tau \Longleftrightarrow \sigma$ means that (in world w) v has type τ iff it has type σ. In contrast, $w \Vdash v : \mathsf{teq}(\tau,\sigma)$ means that *every* value in world w has type τ iff it has type σ; that is, teq is our notion of type equality. In each world, the type $\mathsf{teq}(\tau,\sigma)$ is equal to either top or bot; we call such types *world types* because they depend only on the world, not on the value. Henceforth if τ is a world type we will write $w \Vdash \tau$ to mean $w \Vdash v : \tau$ for an arbitrary v.

Vector values. As we have said, our type system applies to several computational models (e.g., λ-calculus, von Neumann) and in some of these we need to type several kinds of values. We have *locations* $l : \mathrm{Loc}$ that index a mutable store m; *storable values* $u : SV$ that are the range of m (contents of memory cells); and *values* $v : V$. We assume $\mathrm{Loc} \subset SV$.

On a von Neumann machine, $SV = \mathrm{Loc}$ and v is a vector of locations—one could think of a register-bank—indexed by a natural number j. That is, if v is a value then $v(j)$ is a Loc. In order to type locations, we choose an injective function $\overrightarrow{\cdot}$ from storable values to values, for instance $\overrightarrow{u} \stackrel{\text{def}}{=} \lambda j.u$. This way the same set of types can be used for all kinds of values.

In λ-calculus, $SV = V$ is the usual set of values in λ-calculus, we have $\mathrm{Loc} \subsetneq SV$ by syntactic inclusion, and we take $\overrightarrow{u} \stackrel{\text{def}}{=} u$.

Singletons and slots. The singleton type $\mathsf{just}\,u$ is the type of the single storable value u. Since $\mathrm{Loc} \subset SV$ we can also write $\mathsf{just}\,l$ for a location l.

$$\mathsf{just}\,u \stackrel{\text{prim}}{=} \{(w,v) \mid v = \overrightarrow{u}\} \tag{13}$$

The construction $!(\mathsf{just}\,u \Rightarrow \tau)$ asserts that the storable-value u has type τ in the current world, which we write $u \wr \tau$.

$$u \wr \tau \stackrel{\text{syn}}{=} !(\mathsf{just}\,u \Rightarrow \tau) \tag{14}$$

[2] In the context of a definition that depends on a formal parameter $w \in W$, we refer to w as the "current" world.

We can define "$\exists l{:}\mathrm{Loc}.\,\mathsf{just}\,l \wedge \tau(l)$", the type of locations l which satisfy some property $\tau(l)$ (where $\tau(l)$ is a world type).

The type $\mathsf{slot}(j,\tau)$ characterizes values v such that (informally) the jth slot has type τ.

$$\mathsf{slot}(j,\tau) \stackrel{\text{prim}}{=} \{(w,v) \mid w \Vdash v(j) \wr \tau\} \tag{15}$$

To say that register 2 has the value 3 we write $\mathsf{slot}(2,\mathsf{just}\,3)$.

4. Necessity and the modal operator $\triangleright$

In this section we will explain an operator that encapsulates the notion of induction; then we will demonstrate its application to recursive types (Section 5), recursive data structures (Sections 6, 8) and recursive functions and control-flow (Section 11).

Judgments. We introduce a notion of judgment and its semantic interpretation. Given two types σ and τ, we write

$$\sigma \vdash \tau \tag{16}$$

when the type σ is a subset of the type τ, or equivalently, when for every world w and for every value v,

$$w \Vdash v : \sigma \qquad \text{implies} \qquad w \Vdash v : \tau.$$

More generally, we write $\sigma_1, \cdots, \sigma_n \vdash \tau$ to mean $\sigma_1 \wedge \cdots \wedge \sigma_n \vdash \tau$. We write $\vdash \tau$ to mean $\mathsf{top} \vdash \tau$.

It is possible to reason about these judgments using a rich set of inference rules. This rules can be used to derive a larger number of type properties without having to expand the definition of type constructors and reason in the underlying logic. Some of these rules are given in Figures 4–7.

The "later" operator $\triangleright$. In any model $(W, R, \Vdash)$ we require the accessibility relation R to be transitive and *well-founded*:

$$\begin{array}{l} R \text{ well-founded} \stackrel{\text{def}}{=} \\ \quad \text{from any world } w \\ \quad \text{there is no infinite path } w\,R\,w'\,R\,w''\,R\ldots \end{array} \tag{17}$$

Intuitively, $w\,R\,w'$ means that the world w' comes at a strictly later stage than the world w. This leads us to define the "later" operator:

$$\triangleright\tau \stackrel{\text{prim}}{=} \{(w,v) \mid \forall w'.\,wRw' \Rightarrow (w',v) \in \tau\} \tag{18}$$

That is, $w \Vdash v : \triangleright\tau$ precisely when $w' \Vdash v : \tau$ for any world w' coming strictly later than the world w.

The following properties follow easily.

Lemma 4.1.

1. *$\triangleright$ is monotone: if $\sigma \vdash \tau$, then $\triangleright\sigma \vdash \triangleright\tau$.*
2. *$\triangleright$ distributes over intersection: $\triangleright \bigwedge \tau_i = \bigwedge \triangleright\,\tau_i$.*

***The necessity operator* $\Box$.** There is a related operator $\Box$, pronounced "necessarily", which means "now and later" and is defined just that way:

$$\Box\,\tau \stackrel{\text{syn}}{=} \tau \wedge \rhd\,\tau \tag{19}$$

The next proposition shows that the operator $\Box$ is a *closure operator* on types, or what is called a *comonad* in category theory[3]—although in this paper we will not use the toolkit of category theory.

Lemma 4.2.

1. *$\Box$ is monotone: if $\sigma \vdash \tau$, then $\Box\,\sigma \vdash \Box\,\tau$.*
2. *For any type τ, $\Box\,\tau \vdash \tau$.*
3. *If $\Box\,\sigma \vdash \tau$, then $\Box\,\sigma \vdash \Box\,\tau$.*
4. *$\Box$ distributes over intersection: $\Box \bigwedge \tau_i = \bigwedge \Box\ \tau_i$.*

Necessary types. We are interested in types τ such that, once a value v has type τ in some world w, it has type τ in every future world. Such types are called *necessary* types:

$$\tau \text{ necessary} \stackrel{\text{def}}{=} \tau \vdash \rhd\,\tau \tag{20}$$

Not all of our types are necessary. In particular, since the mutable store evolves from one world to the next, it will be possible to construct types (sets of configurations) such that $\tau \vdash \rhd\,\tau$ does not hold. In practice, our datatypes (types for references, code pointers, ...), are necessary. However, some logical constructions on types do not preserve necessity: for instance, the fact that two types τ and τ' are necessary does not imply that the type $\tau \Rightarrow \tau'$ is necessary.

Since necessary types are much easier to reason about, we want a convenient way to construct them. This is precisely what the modal operators $\rhd$ and $\Box$ achieve for us: they transform arbitrary types into necessary types. Let us state formally a few properties relating necessity to these two modal operators:

Lemma 4.3.

1. *For every type τ, the types $\Box\,\tau$ and $\rhd\,\tau$ are necessary,*
2. *A type τ is necessary iff $\tau = \Box\,\tau$, that is, $\vdash \mathsf{teq}(\tau, \Box\,\tau)$.*
3. *The intersection of necessary types is necessary.*

The Löb rule. Our hypothesis that the accessibility relation R is well-founded is reflected in our modal logic by the following induction principle:

Theorem 4.4 (Löb Rule). *For every type τ,*

$$\frac{\rhd\,\tau \vdash \tau}{\vdash \tau} \tag{21}$$

Proof. As R is well-founded, it satisfies the following induction principle:

$$\forall P.(\forall w.(\forall w'.\, wRw' \Rightarrow P(w')) \Rightarrow P(w)) \Rightarrow \forall w.P(w).$$

By taking $P(w) = (w, v) \in \tau$, we get:

$$\forall v.(\forall w.(\forall w'.\, wRw' \Rightarrow (w', v) \in \tau) \Rightarrow (w, v) \in \tau) \Rightarrow \forall w.(w, v) \in \tau.$$

Finally, by distributing the quantification over v, we have:

$$(\forall w.\, v.(\forall w'.\, wRw' \Rightarrow (w', v) \in \tau) \Rightarrow (w, v) \in \tau) \Rightarrow \forall w.\, v.(w, v) \in \tau.$$

This is exactly what the rule 21 means. □

Corollary 4.5 (Generalized Löb Rule).

$$\frac{\sigma, \rhd\,\tau \vdash \tau \qquad \sigma \text{ necessary}}{\sigma \vdash \tau}$$

Proof. By a short logical derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\sigma, \rhd\,\tau \vdash \tau}{\sigma, \rhd(\sigma \wedge (\sigma \Rightarrow \tau)) \vdash \tau}}{\sigma, (\rhd\,\sigma) \wedge \rhd(\sigma \Rightarrow \tau) \vdash \tau}}{\sigma \wedge \rhd\,\sigma, \rhd(\sigma \Rightarrow \tau) \vdash \tau}\ \sigma \text{ necessary}}{\sigma, \rhd(\sigma \Rightarrow \tau) \vdash \tau}}{\rhd(\sigma \Rightarrow \tau) \vdash \sigma \Rightarrow \tau}}{\vdash \sigma \Rightarrow \tau}\ \text{Theorem 4.4}}{\sigma \vdash \tau}$$

□

5. Recursive types

Given a type operator $F \in \text{Type} \to \text{Type}$, we can construct the recursive type $\mathsf{rec}\,F$. As in most models of recursive types, if F is *contractive* then $\mathsf{rec}\,F$ is a fixed point of F. Informally, a contractive function is one such that if τ is approximately equal to σ, then $F\tau$ is more accurately equal to $F\sigma$. We can express contractiveness using the primitives of our type system— in fact, contractiveness and nonexpansiveness[4] have very concise, intuitive definitions.

$$F \text{ contractive} \stackrel{\text{def}}{=} \rhd\,\mathsf{teq}(\tau, \sigma) \vdash \mathsf{teq}(F\tau, F\sigma) \tag{22}$$

$$F \text{ nonexpansive} \stackrel{\text{def}}{=} \Box\,\mathsf{teq}(\tau, \sigma) \vdash \mathsf{teq}(F\tau, F\sigma) \tag{23}$$

The hypothesis $\rhd\,\mathsf{teq}(\tau, \sigma)$ means that (in the current world) τ is perhaps not yet equal to σ, but in every future world it will be. What is it about the future worlds that makes them accept $\tau = \sigma$? The answer is that the world-accessibility relation R is well founded, which informally means that in the current world there is a finite number k of computation

[3] In fact, the $\Box$ operator is the free comonad of the $\rhd$ operator, both understood as functors in the category of types ordered by inclusion. This fact may be proved directly, or deduced from our definition 20 of necessary types, and Section 9.4 of Barr and Wells [BW83].

[4] In a way now standard in type theory [MPS86], nonexpansive type constructors may be composed with contractive constructors to make contractive constructors.

steps remaining. Perhaps we can call a function that, given an argument of type τ would get stuck on the kth step, but given an argument of type σ executes safely for k steps; in this case $w \not\Vdash v : \mathsf{teq}(\tau, \sigma)$. But (by the well-foundedness of R), the next world has $< k$ steps of computation remaining to it, so it is less able to distinguish τ from σ. So F contractive means that if τ is indistinguishable from σ in every future world, then $F(\tau)$ is indistinguishable from $F(\sigma)$ in this world.

On the other hand, F nonexpansive means that if τ is indistinguishable from σ in this world and every future world, then $F(\tau)$ is indistinguishable from $F(\sigma)$ in this world. For example, we will later introduce the constructor ref which is contractive, and offset which is nonexpansive.

Given a type operator F, we define by induction on worlds the fixpoint candidate $\mathsf{rec}\, F$.

$$\begin{array}{l}\mathsf{rec}\, F \stackrel{\text{prim}}{=} \text{the unique set } \sigma \text{ such that} \\ \quad (w, v) \in \sigma \qquad \text{iff} \\ \quad (w, v) \in F(\{(w', v) \mid wRw' \wedge (w', v) \in \sigma\}).\end{array} \tag{24}$$

By the well-foundedness of R, this is defined for all type operators F—in order to find out whether $(w, v) \in \mathsf{rec}\, F$ one only needs to consider configurations $(w', v') \in \mathsf{rec}\, F$ where wRw'. This fixpoint candidate is easily defined as a structural `Fixpoint` function in Coq.

Lemma 5.1. *When F is contractive, the type $\mathsf{rec}\, F$ is a fixpoint of F, that is,*

- $\mathsf{rec}\, F \vdash F(\mathsf{rec}\, F)$ *(unfolding)*
- $F(\mathsf{rec}\, F) \vdash \mathsf{rec}\, F$ *(folding)*

These two properties can also be phrased as a single assertion: $\vdash \mathsf{teq}(\mathsf{rec}\, F, F(\mathsf{rec}\, F))$.

Lemma 5.2. *The fixpoint of a contractive operator F is unique: if $\vdash \mathsf{teq}(\tau, F(\tau))$ and $\vdash \mathsf{teq}(\sigma, F(\sigma))$, then $\vdash \mathsf{teq}(\tau, \sigma)$.*

6. A Kripke semantics of stores

One advantage of our semantic approach is that we did not need to fully specify our Kripke model in order to interpret the logical part of our typing system (intersection and union types, quantification, necessity, recursive types). But since we are particularly interested here in languages with mutable references—we would like to include a notion of *reference type* in our system—this requires us to explain what Kripke model of worlds w and accessibility relation wRw' suitable for mutable references we have in mind.

Recall that, intuitively, the current world w shall be used for two purposes: (1) to ensure that the accessibility relation R is well founded, and (2) to constrain what values may be contained in memory locations. That is, a world $w = (n, \Psi)$ will be a pair of an index $n \in \mathbb{N}$ (which counts down as time advances) and a store typing Ψ. As one should expect, this store typing Ψ will be a partial function from memory locations $l \in \mathrm{Loc}$ to types. However, we will have to be extremely careful, and restrict ourselves to only some of these functions. Indeed, the mutually recursive equations in W and Type below have no solution (by a cardinality argument):

$$\begin{array}{c} W = \mathbb{N} \times (\mathrm{Loc} \rightharpoonup \mathrm{Type}) \\ \mathrm{Type} = \mathcal{P}\left(W \times V\right). \end{array} \tag{25}$$

The significance of $\triangleright$. Let us carry on our analysis. Informally, a memory $m \in \mathrm{Mem}$ is well typed with respect to a store typing Ψ if for all locations l in the domain of Ψ, the value $m(l)$ has type $\Psi(l)$. Let us think ahead and imagine that the set of worlds W is already constructed. One could then define the world type $l \mapsto \tau$ which characterizes the worlds (n, Ψ) in which the memory-typing for l is type τ:

$$l \mapsto \tau \stackrel{\text{prim}}{=} \{((n, \Psi), v) \mid \Psi(l) = \tau\} \tag{26}$$

Then, we may declare that a memory m is well-typed when the following assertion holds:

$$\begin{array}{l}\mathsf{validmem}(m) \stackrel{\text{syn}}{=} \\ \quad \forall l{:}\mathrm{Loc}.\forall \tau{:}\mathrm{Type}.\, l \mapsto \tau \Rightarrow \triangleright(m(l) \,{:}\, \tau)\end{array} \tag{27}$$

In this definition we write $\triangleright(m(l) \,{:}\, \tau)$. There is some value u in memory at address l, and we guarantee to every future world that $u \,{:}\, \tau$. We don't need to guarantee $u \,{:}\, \tau$ in the *current* world because it takes one step just to dereference $m(l)$, and in that step we move to a future world.

This use of the $\triangleright$ operator rather than the $\Box$ operator is crucial in order to solve the cardinality issue. Indeed, for a configuration $((n, \Psi), v)$, only the configurations of index strictly less than n are then relevant in the type $\Psi(l)$.

The Kripke semantics. This informal discussion enables us to stratify our definition of worlds and types, and to start by defining "finitely stratified" worlds and types of rank n by mutual induction:

$$\begin{array}{c} W_n \stackrel{\text{def}}{=} \{n\} \times (\mathrm{Loc} \rightharpoonup \mathrm{Type}_n) \\ \mathrm{Type}_n \stackrel{\text{def}}{=} \mathcal{P}\left(\bigcup_{k<n} W_k \times V\right) \end{array} \tag{28}$$

The set of worlds is then defined as the union of finitely stratified worlds:

$$W \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} W_n.$$

Finally, the set of types is defined[5] according to the principles of Kripke semantics presented in Section 2:

$$\mathrm{Type} \stackrel{\text{def}}{=} \mathcal{P}\left(W \times V\right).$$

[5] **Remark.** The set Type is isomorphic to the set $\prod_{n \in \mathbb{N}} \mathcal{P}\left(W_n \times V\right)$ obtained as projective limit of the sequence of finitely stratified types $\{\mathrm{Type}_n\}_{n \in \mathbb{N}}$, where $\mathrm{Type}_n \cong \prod_{k<n} \mathcal{P}\left(W_k \times V\right)$.

We have the inclusions

$$\mathrm{Type}_0 \subset \ldots \subset \mathrm{Type}_n \subset \ldots \subset \mathrm{Type}$$

and a family of approximation functions

$$\lfloor \_ \rfloor_n : \mathrm{Type} \to \mathrm{Type}_n$$

defined thus:

$$\lfloor \tau \rfloor_n \stackrel{\mathrm{def}}{=} \{((k, \Psi), v) \in \tau \mid k < n\}. \tag{29}$$

These approximation functions are lifted pointwise to store typings Ψ such that

$$\begin{aligned} \mathrm{dom} \lfloor \Psi \rfloor_n &= \mathrm{dom}\, \Psi, \\ \lfloor \Psi \rfloor_n(l) &= \lfloor \Psi(l) \rfloor_n. \end{aligned} \tag{30}$$

Finally, the accessibility relation between worlds is defined as follows:

$$\begin{aligned} &(n, \Psi)\, R\, (n', \Psi') \stackrel{\mathrm{def}}{=} \\ &\quad n > n' \ \wedge\ \forall l \in \mathrm{dom}\, \Psi.\ \Psi'(l) = \lfloor \Psi(l) \rfloor_{n'}. \end{aligned} \tag{31}$$

Intuitively, when moving to a future world, at least one step must be taken; the types of allocated locations must be approximately preserved; and new locations may be allocated.

This completes our definition of a Kripke model for mutable references. The two types $l \mapsto \tau$ and $\mathsf{validmem}(m)$ which were used for earlier discussion, are then defined as in equations 26 and 27.

Reference types. We are ready now to define reference types. Intuitively, a reference of type τ is a pointer to a location l of type τ. Because of the inherent stratification of store typing, this is actually too strong a requirement. We need to replace it by the weaker but sufficient requirement that the type τ' of the location l coincides with the type τ in the future. Hence the following definition of reference types:

$$\mathsf{ref}\, \tau \stackrel{\mathrm{syn}}{=} \exists l{:}\mathrm{Loc}.\, \mathsf{just}\, l \wedge \exists \tau'{:}\mathrm{Type}.(l \mapsto \tau' \wedge \triangleright\, \mathsf{teq}(\tau, \tau')) \tag{32}$$

Note that this definition is given solely in terms of type constructors already defined. We have been arguing for the usefulness of a modal approach to type semantics, as well as a finer-grained decomposition of such semantics; definition 32 exemplifies both of these ideas in a nice way. To underscore the flexibility of this decomposition, we speculate that one could synthesize, without dipping back into the underlying logic, a replacement for "$l \mapsto \tau'$" that would yield a region-enabled definition of the ref constructor, and thereby permit deallocation.

7. Type preservation

Suppose $v : \tau$ in some state, and then a memory location is updated—either to initialize a new reference or to update an old one. One of the usual difficulties in a semantics of references is to ensure that $v : \tau$ in the new state. For example, if $\tau = \mathsf{list}(\mathsf{ref}\ \sigma)$ (with list defined in equation 61, below) how can we ensure that the store to memory has not modified one of the list cells or one of the references? In our framework the solution is (1) use R to constrain the evolution of worlds; (2) make sure that τ is *necessary*, that is, τ is preserved as we change worlds; and (3) use $\mathsf{validmem}()$ to constrain the machine *states* to follow the evolution of *worlds*. Concretely, we demonstrate that, with our choice of accessibility relation, reference types are necessary and all interesting store operations (memory access, allocation and update) are permitted.

We use the following technical lemma several times.

Lemma 7.1 (Approximation and Equality). *A type and its approximation can be related as follows.*

$$(k, \Psi) \Vdash \triangleright\, \mathsf{teq}(\tau, \lfloor \tau \rfloor_k)$$

We show that an allocated memory location remains allocated forever and that its type is preserved.

Lemma 7.2 (Store Typing Preservation).

$$l \mapsto \tau \vdash \triangleright\, \exists \sigma{:}\mathrm{Type}.l \mapsto \sigma \tag{33}$$

$$l \mapsto \tau \vdash \triangleright\, \forall \sigma{:}\mathrm{Type}.l \mapsto \sigma \Rightarrow \triangleright\, \mathsf{teq}(\tau, \sigma) \tag{34}$$

Proof. We assume $(k, \Psi) \Vdash l \mapsto \tau$, that is, $\Psi(l) = \tau$. Let (k', Ψ') be a world such that $(k, \Psi) R (k', \Psi')$.

By definition 31, we have $\mathrm{dom}\, \Psi \subset \mathrm{dom}\, \Psi'$, and therefore, for some type σ, we have $\Psi'(l) = \sigma$, that is, $(k', \Psi') \Vdash l \mapsto \sigma$. This proves the first assertion.

Now, let σ be a type such that $(k', \Psi') \Vdash l \mapsto \sigma$, that is, $\Psi'(l) = \sigma$. By definition 31, $\sigma = \lfloor \tau \rfloor_{k'}$. Hence, by lemma 7.1, $(k', \Psi') \Vdash \triangleright\, \mathsf{teq}(\tau, \sigma)$ as required to prove the second assertion. □

In particular, if a storable value u can be stored at location l now, it can still be stored at this location in the future:

$$l \mapsto \tau \wedge \triangleright(u \circledcirc \tau) \vdash \triangleright\, \forall \sigma{:}\mathrm{Type}.l \mapsto \sigma \Rightarrow \triangleright(u \circledcirc \sigma). \tag{35}$$

Another direct corollary of the lemma is that reference types are necessary.

We now show that the expected store operations are permitted. For instance, if the location l has type $\mathsf{ref}\, \tau$, the storable-value u has type τ and the current memory m is valid, we should be able to store u at location l; that is, the memory $m[l := u]$ should be valid in some world one step after the current world.

Lemma 7.3 (Unchanged memory). *If we have*

$$(k + 1, \Psi) \Vdash \mathsf{validmem}(m) \tag{36}$$

then there exists a store typing Ψ' such that

$$(k + 1, \Psi) R (k, \Psi') \tag{37}$$

$$(k, \Psi') \Vdash \mathsf{validmem}(m) \tag{38}$$

$$\mathrm{dom}\, \Psi' = \mathrm{dom}\, \Psi \tag{39}$$

Proof. We take $\Psi' = \lfloor \Psi \rfloor_k$. We clearly have assertions 37 and 39. We now prove assertion 38 using definition 27. Let l be a location and τ be a type such that $(k, \Psi') \Vdash l \mapsto \tau$. By definition 26, as $\mathrm{dom}\,\Psi' = \mathrm{dom}\,\Psi$, there exists τ' such that $(k+1, \Psi) \Vdash l \mapsto \tau'$. Furthermore, by assertion 36, $(k+1, \Psi) \Vdash \triangleright(m(l) \colon\!\!\colon \tau')$. Hence, by assertion 35, $(k, \Psi') \Vdash \triangleright(m(l) \colon\!\!\colon \tau)$ as wanted. □

Lemma 7.4 (Memory update). *If the type τ is necessary and*

$$(k+1, \Psi) \Vdash \mathsf{validmem}(m) \tag{40}$$
$$(k+1, \Psi) \Vdash (l \colon\!\!\colon \mathsf{ref}\,\tau) \tag{41}$$
$$(k+1, \Psi) \Vdash (u \colon\!\!\colon \tau) \tag{42}$$

then there exists a store typing Ψ' such that

$$(k+1, \Psi) R (k, \Psi') \tag{43}$$
$$(k, \Psi') \Vdash \mathsf{validmem}(m[l := u]) \tag{44}$$
$$\mathrm{dom}\,\Psi' = \mathrm{dom}\,\Psi \tag{45}$$

Proof. We take $\Psi' = \lfloor \Psi \rfloor_k$ as for the previous lemma 7.3. This ensures that $(k+1, \Psi) R (k, \Psi')$ and $\mathrm{dom}\,\Psi' = \mathrm{dom}\,\Psi$. We now prove assertion 44 using definition 27. Let l' be a location and τ' be a type such that $(k, \Psi') \Vdash l' \mapsto \tau'$. If $l' \neq l$, we can conclude as in lemma 7.3. So, we suppose that $l' = l$. From assumption 41 and by definition 32, there exists a type τ'' such that $(k+1, \Psi) \Vdash l \mapsto \tau''$ and $(k+1, \Psi) \Vdash \triangleright\, \mathsf{teq}(\tau, \tau'')$. From assumption 42, using the fact that τ is necessary, we therefore get $(k+1, \Psi) \Vdash \triangleright(u \colon\!\!\colon \tau'')$. Hence, by assertion 35, $(k, \Psi') \Vdash \triangleright(u \colon\!\!\colon \tau')$ as wanted. □

Lemma 7.5 (Memory allocation). *If the type τ is necessary, and*

$$(k+1, \Psi) \Vdash \mathsf{validmem}(m) \tag{46}$$
$$(k+1, \Psi) \Vdash (u \colon\!\!\colon \tau) \tag{47}$$
$$l \notin \mathrm{dom}\,\Psi \tag{48}$$

then there exists a store typing Ψ' such that

$$(k+1, \Psi) R (k, \Psi') \tag{49}$$
$$(k, \Psi') \Vdash \mathsf{validmem}(m[l := u]) \tag{50}$$
$$(k, \Psi') \Vdash (l \colon\!\!\colon \mathsf{ref}\,\tau) \tag{51}$$
$$\mathrm{dom}\,\Psi' = \mathrm{dom}\,\Psi \cup \{l\} \tag{52}$$

Proof. We take $\Psi' = \lfloor \Psi \rfloor_k \cup \{l \mapsto \lfloor \tau \rfloor_k\}$. This is well-defined as $l \notin \mathrm{dom}\,\Psi'$. Clearly, we have $(k+1, \Psi) R (k, \Psi')$ and $\mathrm{dom}\,\Psi' = \mathrm{dom}\,\Psi \cup \{l\}$. We now prove assertion 50 using definition 27. Let l' be a location and τ' be a type such that $(k, \Psi') \Vdash l' \mapsto \tau'$. If $l' \neq l$, we can conclude as in lemma 7.3. So, we suppose that $l' = l$. From assumption 47, using the fact that τ is necessary, we get $(k, \Psi') \Vdash \triangleright(u \colon\!\!\colon \tau)$. Using lemma 7.1, this gives $(k, \Psi') \Vdash \triangleright(u \colon\!\!\colon \lfloor \tau \rfloor_k)$. Thus, using the fact that $\tau' = \Psi'(l) = \lfloor \tau \rfloor_k$, we have $(k, \Psi') \Vdash \triangleright(u \colon\!\!\colon \tau')$ as wanted. Finally, by lemma 7.1, $(k, \Psi') \Vdash \triangleright\, \mathsf{teq}(\tau, \lfloor \tau \rfloor_k)$, and by definition of Ψ', $(k, \Psi') \Vdash l \mapsto \lfloor \tau \rfloor_k$. Therefore, we have assertion 51 by definition 32. □

Lemma 7.6 (Memory access). *If*

$$w \Vdash (l \colon\!\!\colon \mathsf{ref}\,\tau) \tag{53}$$
$$w \Vdash \mathsf{validmem}(m) \tag{54}$$
$$w R w' \tag{55}$$

then

$$w' \Vdash (m(l) \colon\!\!\colon \tau) \tag{56}$$

Proof. From assumption 53 there exists a type τ' such that $w \Vdash l \mapsto \tau'$ and $w \Vdash \triangleright\, \mathsf{teq}(\tau, \tau')$. Using assumption 54 we get $w \Vdash \triangleright\, m(l) \colon\!\!\colon \tau'$ and then $w \Vdash \triangleright\, m(l) \colon\!\!\colon \tau$. Finally, by definition of $\triangleright$, $w' \Vdash (m(l) \colon\!\!\colon \tau)$, as wanted. □

With these lemmas, we can prove soundness of rules for type-checking store instructions for initializing new references or updating old ones (Appendix A).

8. Data structures

To describe the memory layout of data structures and of the stored program, we need address arithmetic on locations (Loc). In this paper we will identify Loc with the integers. To reason about address arithmetic on Loc, we have arithmetic operators in the type system:

$$\mathsf{plus}(i, j, k) \stackrel{\text{prim}}{=} \{(w, v) \mid i + j = k\} \tag{57}$$
$$\mathsf{greater}(i, j) \stackrel{\text{prim}}{=} \{(w, v) \mid i > j\} \tag{58}$$

From these we can construct the offset type constructor, such that $l \colon\!\!\colon \mathsf{offset}(n, \tau)$ iff $l + n \colon\!\!\colon \tau$. This is useful for record fields and program labels.

$$\mathsf{offset}(n, \tau) \stackrel{\text{syn}}{=} \exists l{:}\mathrm{Loc}.\ \mathsf{just}\, l \wedge \exists l'{:}\mathrm{Loc}.\ \mathsf{plus}(l, n, l') \wedge l' \colon\!\!\colon \tau \tag{59}$$

Now records are just intersections of fields, and disjoint union types such as lists are easy to construct.

$$\mathsf{pair}(\sigma, \tau) \stackrel{\text{syn}}{=} \mathsf{offset}(0, \mathsf{ref}\ \sigma) \wedge \mathsf{offset}(1, \mathsf{ref}\ \tau) \tag{60}$$

$$\mathsf{list}\,\tau \stackrel{\text{syn}}{=} \mathsf{rec}\,\alpha.\ \mathsf{just}\, 0 \vee \exists l{:}\mathrm{Loc}.\ \mathsf{just}\, l \wedge \mathsf{greater}(l, 0) \wedge \mathsf{pair}(\tau, \alpha) \tag{61}$$

That is, nil of a list is zero; the cons cell of a list must be at an address l, such that $l > 0$, and also must be a pair of an element τ and a list α.

Immutable references. An immutable reference to type τ can be modeled as a mutable reference to a singleton l, such that (in this world) l has type τ:

$$\mathsf{iref}\ \tau \stackrel{\text{syn}}{=} \exists l{:}\mathrm{Loc}.\, \mathsf{ref}(\mathsf{just}\ l) \wedge l \colon\!\!\colon \tau \tag{62}$$

Henry Ford sold Model Ts in any color the customer wanted, so long as it was black. An immutable ref is a mutable ref into which the customer can store any value he wants, so long as it's the value that's already there.

9. Safe states in safe worlds

We wish to reason about function values and function application. A function of type $\tau \to \sigma$ (or, on a von Neumann machine, a code pointer of type $\mathsf{codeptr}(\tau)$) is a value that is *safe* to apply to arguments of type τ. To synthesize function or code-pointer types, we must first model *safety* in our type system.

Our notion of "safe for k steps" is independent of our choice of calculus. It requires only that the calculus comes equipped with a small-step relation $\rightarrowtail \subset \mathrm{State} \times \mathrm{State}$.

$$\mathrm{safe}_0(s) \stackrel{\mathrm{def}}{=} \mathrm{True} \tag{63}$$

$$\mathrm{safe}_{k+1}(s) \stackrel{\mathrm{def}}{=} (\exists s'.\, s \rightarrowtail s') \wedge \forall s'.\, s \rightarrowtail s' \Rightarrow \mathrm{safe}_k\, s' \tag{64}$$

For a calculus of mutable references, we assume that $\mathrm{State} = V \times \mathrm{Mem}$, that is, the state (v, m) on a von Neumann machine has register-bank v and memory m, or in a λ-calculus has expression v and store m.

The type $\mathsf{safemem}(m)$ expresses the set of configurations (w, v) in which it is safe to run forward from state (v, m). Note that the world w contains an "expiration date" n telling how many steps forward we may attempt to take. After n steps we declare victory, that is, we do not care whether the small-step relation might get stuck *after* the nth step.

$$\mathsf{safemem}(m) \stackrel{\mathrm{prim}}{=} \{((n, \Psi), v) \mid \mathrm{safe}_n(v, m)\} \tag{65}$$

$$\mathsf{safe} \stackrel{\mathrm{syn}}{=} \forall m{:}\mathrm{Mem}.\mathsf{validmem}(m) \Rightarrow \mathsf{safemem}(m) \tag{66}$$

The judgment $w \Vdash v : \mathsf{safe}$ is an important one. A memory m is "in" a world w whenever $w \Vdash \mathsf{validmem}(m)$. A value v is safe in w when, for all m "in" w, the state (v, m) is safe for all the steps remaining in w. It is on the primitive constructor safe that we will build (in Section 11) the notion of $\mathsf{codeptr}$, that is, continuation-pointer.

10. λ-calculus

The reader may skip this section without loss of comprehension of later sections. The main point of departure here is Ahmed's thesis [Ahm04], especially Section 3.3.5, which gives constructions used in defining the type constructor for arrow types.

Our type system works well to classify the terms of systems of λ-calculus, such as the one given below. In fact, all of our previous definitions apply without modification; recall that for λ-calculus we have $SV = V$ and $\overrightarrow{u} \stackrel{\mathrm{def}}{=} u$.

$$e ::= () \mid x \mid l \mid \lambda x.\, e \mid (e_1 e_2) \mid \mathsf{new}\, e \mid !\, e \mid e_1 := e_2 \tag{67}$$

$$u, v ::= () \mid l \mid \lambda x.\, e \tag{68}$$

This is a λ-calculus extended with mutable references. It's a contribution of our system that even in the presence of such references, impredicative type abstraction and type application (and so forth) are coercions with no operational significance; in particular, terms of quantified type have no explicit introduction or elimination forms, such as "pack" and "unpack". Section 13.1 discusses how we achieve this.

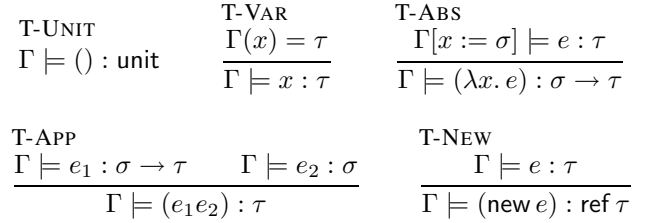

Figure 3. Typing rules specific to λ-calculus

Because we define our types semantically, we decline to give an inductive definition for type expressions. Moreover we treat λ-calculus à la Curry and omit type decorations from our terms. We use the standard small-step call-by-value operational semantics, and omit the rules for same.

Definitions of the primitive type constructors, and those for quantified, recursive, and reference types remain unchanged. It therefore remains to give semantics to the unit and arrow types; the typing judgments for closed terms, substitutions, and open terms; and to prove the typing lemmas (which would be rules in a syntactic progress-and-preservation proof) for abstraction and application.

First the typing judgment for closed terms.

$$\begin{aligned} e :_{(k,\Psi)} \tau \stackrel{\mathrm{def}}{=}\ & \forall j < k, m, m', e'. \\ & (k, \Psi) \Vdash \mathsf{validmem}(m) \\ & \Rightarrow (m, e) \rightarrowtail^j (m', e') \ \wedge\ \mathrm{irred}(m', e') \\ & \Rightarrow \exists \Psi'.\ (k, \Psi) R^{=} (k - j, \Psi') \\ & \qquad \wedge (k - j, \Psi') \Vdash \mathsf{validmem}(m') \\ & \qquad \wedge (k - j, \Psi') \Vdash e' : \tau \end{aligned} \tag{69}$$

The state (m, e) may step to (m', e') which is irreducible (either a value or *stuck*); but since $\_ \Vdash e' : \tau$ only when e' is a value, then a well-typed term e cannot get stuck within k steps.

There are two new type constructors of interest,

$$\mathsf{unit} \stackrel{\mathrm{prim}}{=} \{(w, ()) \mid \mathrm{True}\}, \text{ and} \tag{70}$$

$$\tau \to \sigma \stackrel{\mathrm{syn}}{=} \triangleright(\tau \to' \sigma), \text{ where} \tag{71}$$

$$\tau \to' \sigma \stackrel{\mathrm{prim}}{=} \{(w, \lambda x.\, e) \mid \forall w, v.(w, v) \in \tau \Rightarrow e[v/x] :_w \sigma\}.$$

Finally we define the general typing judgment $\models$ in terms of ground substitutions γ:

$$\gamma :_w \Gamma \stackrel{\text{def}}{=} \forall x \in \operatorname{dom}\Gamma.\, \gamma(x) :_w \Gamma(x)$$
$$\Gamma \models e : \tau \stackrel{\text{def}}{=} \forall w, \gamma.\, \gamma :_w \Gamma \Rightarrow \gamma(e) :_w \tau$$

The expected typing rules (Figure 3) can be proved as lemmas from the model.

One of our major semantic innovations is the decomposition of function types into primitive notions such as safety. Equation 69 does not show this, because so far we have worked out this more modular decomposition only in the von Neumann setting. Therefore the next section explains this decomposition in terms of code pointers.

11. Machine instructions

Sections 2–9 apply to models of computation ranging from λ-calculus to von Neumann machines. Now we will discuss von Neumann machines specifically. A location of type $\mathsf{codeptr}(\tau)$ is the address of a continuation to which it's safe to jump, providing one passes an argument of type τ. That is, if $l \,\mathring{:}\, \mathsf{codeptr}(\tau)$ then one can safely set the program counter to l, as long as the registers satisfy τ.

$$\begin{array}{l}\mathsf{codeptr}(\tau) \stackrel{\text{syn}}{=} \\ \quad \exists l{:}\mathrm{Loc}.\, \mathsf{just}\, l \wedge \triangleright !(\mathsf{slot}(\mathsf{pc}, \mathsf{just}\, l) \wedge \tau \Rightarrow \mathsf{safe})\end{array} \tag{72}$$

At any time strictly in the future, whenever it is the case that the program counter $= l$ and the (other) registers satisfy τ, then the state must be safe.[6]

A machine-instruction ι is a relation on machine states, $\iota \subset \mathrm{State} \times \mathrm{State}$. If the program counter in state s_1 points to a memory location containing a number n that codes for ι, then the machine will step to a state s_2 such that $\iota(s_1, s_2)$.

An example of such an instruction is $r3 \leftarrow m(r2 + 9)$, which we can write as

$$\begin{array}{l}\textsc{load}(3,2,9) = \{((v_1,m_1),(v_2,m_2)) \mid \\ \quad v_2(\mathsf{pc}) = v_1(\mathsf{pc}) + 1 \wedge v_2(3) = m_1(v_1(2)+9) \wedge \\ \quad \forall i \notin \{3, \mathsf{pc}\}.\, v_2(i) = v_1(i) \wedge m_1 = m_2\}\end{array}$$

A number n codes for an instruction ι just when

$$\begin{array}{l}n \text{ encodes } \iota \stackrel{\text{def}}{=} \forall v_1, m_1, v_2, m_2. \\ \quad m_1(v_1(\mathsf{pc})) = n \;\Rightarrow \\ \quad (v_1, m_1) \rightarrowtail (v_2, m_2) \;\Leftrightarrow\; \iota((v_1,m_1),(v_2,m_2))\end{array} \tag{73}$$

Programs. A von Neumann program p is a sequence of integers $p(0), \ldots, p(n-1)$ such that each $p(i)$ encodes some instruction ι_i. For example, perhaps on some machine 7320 encodes $\textsc{load}(3,2,0)$, we might have a program $p = (7320, 4231, 6007)$:

Machine inst.	ι	*Pseudocode*
7320	$\textsc{load}(3,2,0)$	$r3 \leftarrow m(r2+0)$
4231	$\textsc{addimm}(2,3,1)$	$r2 \leftarrow r3+1$
6007	$\textsc{jump}(7)$	$\mathsf{pc} \leftarrow r7$

(74)

Program-code types. For any program p, define

$$\Delta_p \stackrel{\text{syn}}{=} \bigwedge_{i \in \operatorname{dom} p} \mathsf{offset}(i, \mathsf{ref}(\mathsf{just}\ p(i))) \tag{75}$$

(The domain of p is finite; the $\wedge_{i\in\operatorname{dom} p}$ is meta-notation for an expression with $|\operatorname{dom} p|$ conjuncts.) For example, our sample program p (equation 74) has,

$$\begin{array}{rl}\Delta_0 = & \mathsf{offset}(0, \mathsf{ref}(\mathsf{just}\ 7320)) \\ \Delta_1 = & \mathsf{offset}(1, \mathsf{ref}(\mathsf{just}\ 4231)) \\ \Delta_2 = & \mathsf{offset}(2, \mathsf{ref}(\mathsf{just}\ 6007)) \\ \Delta_p = & \Delta_0 \wedge \Delta_1 \wedge \Delta_2\end{array}$$

Type-checking the example program p**.** Suppose that on entry to location l, register $r2$ is a $\mathsf{ref}\ \mathsf{int}$ and that register $r7$ is a $\mathsf{codeptr}(\mathsf{slot}(2, \mathsf{int}))$, that is, a continuation that requires an integer in register 2. Then the program dereferences the ref, adds 1, and continues by jumping to the address in $r7$. We can explain this as $l : \Gamma$ where

$$\Gamma = \mathsf{codeptr}(\mathsf{slot}(2, \mathsf{ref}\ \mathsf{int}) \wedge \mathsf{slot}(7, \mathsf{codeptr}(\mathsf{slot}(2, \mathsf{int}))))$$

Now we wish to prove that, in any world where this program is loaded at address l, then l is also a continuation as described by Γ. We write this claim as,

$$\Delta_p \vdash \Gamma$$

Tan [TA06, Tan05] gives a compositional logic for control flow in which such statements can be proved. He proves soundness of this logic using the indexed model. Our primitives can express Tan's semantics in a clean and simple way.

The form of an instruction-typing rule is $\Delta, \triangleright\Gamma \vdash \Gamma'$ where Δ describes an instruction in memory at location l, Γ describes the type of the continuation at location $l+1$, and the conclusion Γ' claims the type of the continuation at l. Consider a $\textsc{load}(i,j,k)$ at location l. A sound but naive typing rule is,

$$\frac{n \text{ encodes } \textsc{load}(i,j,k) \quad i \neq \mathsf{pc} \quad \tau \text{ necessary}}{\begin{array}{l}\mathsf{ref}(\mathsf{just}\, n),\ \triangleright\, \mathsf{offset}(1, \mathsf{codeptr}(\mathsf{slot}(i, \tau))) \vdash \\ \quad \mathsf{codeptr}(\mathsf{slot}(j, \mathsf{offset}(k, \mathsf{ref}\, \tau)))\end{array}} \tag{76}$$

That is, if the load instruction is at address l, and it's safe to continue at $l+1$ provided that register i has a value of type τ, then it's also safe to continue at l provided that $r_j + k$ has a value of type $\mathsf{ref}\, \tau$.

This is insufficiently general: it neglects the fact that all the registers other than i are not changed by the load

[6] Why "strictly in the future"—why not right now? The codeptr does not require $\Box$ instead of $\triangleright$ for two reasons: on the one hand, then it would not be contractive; and on the other hand, executing a jump to location l takes at least one step, so we don't need the stronger $\Box$ condition. Contractiveness is a useful property for codeptr, since it allows the recursive type $\mathsf{rec}\ \alpha.\ \mathsf{codeptr}(\alpha)$.

instruction. Suppose we have a type ϕ that characterizes the type of several registers but leaves i unrestricted. We say, "i is not in the domain of ϕ," that is,

$$\begin{array}{l}\mathrm{notindom}(i,\phi) \stackrel{\mathrm{def}}{=} \\ \quad \forall w,v,u.(w,v) \in \phi \Rightarrow (w, v[i{:=}u]) \in \phi\end{array} \tag{77}$$

A stronger load rule is,

$$\frac{\begin{array}{cc} n \text{ encodes } \textsc{load}(i,j,k) & i \neq \mathtt{pc} \\ \tau \text{ necessary} & \phi \text{ necessary} \\ \mathrm{notindom}(i,\phi) & \mathrm{notindom}(\mathtt{pc},\phi)\end{array}}{\begin{array}{l}\mathsf{ref}(\mathsf{just}\, n),\ \rhd \mathsf{offset}(1, \mathsf{codeptr}(\phi \wedge \mathsf{slot}(i,\tau))) \vdash \\ \quad \mathsf{codeptr}(\phi \wedge \mathsf{slot}(j, \mathsf{offset}(k, \mathsf{ref}\,\tau)))\end{array}} \tag{78}$$

With respect to our sample program p (equation 74) we can write the invariants for locations $l+0$ through $l+2$ as,

$$\begin{array}{ll}\Gamma_0 = & \mathsf{offset}(0, \mathsf{codeptr}(\phi \wedge \mathsf{slot}(2, \mathsf{ref\,int}))) \\ \Gamma_1 = & \mathsf{offset}(1, \mathsf{codeptr}(\phi \wedge \mathsf{slot}(3, \mathsf{int}))) \\ \Gamma_2 = & \mathsf{offset}(2, \mathsf{codeptr}(\phi \wedge \mathsf{slot}(2, \mathsf{int}))) \\ \Gamma_p = & \Gamma_0 \wedge \Gamma_1 \wedge \Gamma_2 \\ \multicolumn{2}{l}{\text{where } \phi = \mathsf{slot}(7, \mathsf{codeptr}(\mathsf{slot}(2, \mathsf{int})))}\end{array} \tag{79}$$

Then we can use rules for LOAD, ADDIMM, and JUMP to show,

$$\begin{array}{lll}\Delta_0,\ \rhd\Gamma_1 & \vdash & \Gamma_0 \\ \Delta_1,\ \rhd\Gamma_2 & \vdash & \Gamma_1 \\ \Delta_2 & \vdash & \Gamma_2\end{array} \tag{80}$$

11.1 Whole-program typing

The example program is not recursive and does not loop, so in our example, the proof for each instruction i needs only $\rhd\Gamma_{i+1}$. However, our type system is applicable to programs with loops and recursion. Suppose i is a conditional branch to location j, then we need,

$$\Delta_i,\ \rhd(\Gamma_j \wedge \Gamma_{i+1}) \vdash \Gamma_i.$$

The general case is $\Delta_i, \rhd\Gamma_p \vdash \Gamma_i$.

Tan's calculus for control flow [TA06, Tan05] manages such proofs in a modular way. Given proofs of this form for each i, we can combine them using rules for type intersection into,

$$\Delta_p,\ \rhd\Gamma_p \vdash \Gamma_p \tag{81}$$

So, in general, we have a program p and its associated Δ_p. An oracle (perhaps a type-preserving compiler) provides a hypothesis Γ_p giving a codeptr type for each address. An automatic type-checker uses our rules—or syntax-directed specializations of our rules, as described by Wu [Wu05]—to prove $\Delta_p, \rhd\Gamma_p \vdash \Gamma_p$. All ref types are necessary, and necessity is preserved by offset and intersection, therefore Δ_p is necessary. Using the generalized Löb rule (Corollary 4.5), we deduce that

$$\Delta_p \vdash \Gamma_p \tag{82}$$

This means, "in any world where the program code described by Δ_p is loaded, the loop-invariants Γ_p hold."

The initial entry point. Let us assume that the "beginning of the program" is at offset 0 in a complete program, and suppose the hypothesis for offset 0 is $\mathsf{codeptr}(\mathsf{top})$, meaning that it is (claimed) safe to jump to the beginning of the program regardless of the contents of the registers. Then by the rules of type intersection we have $\Gamma_p \vdash \mathsf{codeptr}(\mathsf{top})$. Consequently (by equation 82) we have

$$\Delta_p \vdash \mathsf{codeptr}(\mathsf{top}). \tag{83}$$

Now we will show that this implies the program is safe.

12. Soundness

A von Neumann program is *loaded at* l in a machine-state (r,m) if $r(\mathtt{pc}) = l$ and for each $i \in \mathrm{dom}\, p$, $m(l+i) = p(i)$.

Lemma 12.1. *If program p is loaded at l in machine-state (v,m), then for any natural number k one can construct a world (k,Ψ) such that,*

$$(k,\Psi) \Vdash \mathsf{validmem}(m) \tag{84}$$

$$(k,\Psi) \Vdash v : \mathsf{slot}(\mathtt{pc}, \Delta_p) \tag{85}$$

Proof. The type constructors used in Δ_p are very simple: just intersection, ref, and just. Therefore we can use a simple construction of Ψ:

$$\Psi = \{(l+i) \mapsto \lfloor \mathsf{just}\, p(i) \rfloor_k \mid i \in \mathrm{dom}\, p\}$$

We first check equation 84, that is, by equation 27, if $(k,\Psi) \Vdash l' \mapsto \tau$, then $(k,\Psi) \Vdash \rhd(m(l') \wr \tau)$. By definition 26, when the hypothesis holds, there exists $i \in \mathrm{dom}\, p$ such that $l' = l + i$ and $\tau = \lfloor \mathsf{just}\,(p(i)) \rfloor_k$. The conclusion then becomes $(k,\Psi) \Vdash \rhd(m(l+i) \wr \lfloor \mathsf{just}\,(p(i)) \rfloor_k)$, that is, $(k,\Psi) \Vdash \rhd(\mathsf{just}\,(p(i)) \Rightarrow \lfloor \mathsf{just}\,(p(i)) \rfloor_k)$. This is a consequence of lemma 7.1.

We now consider equation 85. Let $i \in \mathrm{dom}\, p$ and v' be a value. We have

$$\begin{array}{l}(k,\Psi) \Vdash l + i \mapsto \lfloor \mathsf{just}\, p(i) \rfloor_k \\ (k,\Psi) \Vdash \rhd \mathsf{teq}(\mathsf{just}\, p(i), \lfloor \mathsf{just}\, p(i) \rfloor_k).\end{array}$$

by definition 26 and lemma 7.1. Therefore, by equation 32, $(k,\Psi) \Vdash l + i \wr \mathsf{ref}\,(\mathsf{just}\, p(i))$. Hence, by equation 59, $(k,\Psi) \Vdash l \wr \mathsf{offset}(i, \mathsf{ref}\,(\mathsf{just}\, p(i)))$ for all $i \in \mathrm{dom}\, p$. By intersection, we get $(k,\Psi) \Vdash l \wr \Delta_p$. We conclude by equation 15. □

Theorem 12.2 (Soundness). *If program p is loaded at l in machine-state (v,m), and $\Delta_p \vdash \mathsf{codeptr}(\mathsf{top})$, then state (v,m) is safe (for any number of steps k).*

Proof. By Lemma 12.1, construct the world $w = (k+1, \Psi)$. By Lemma 7.3, there exists a world $w' = (k, \Psi')$ such that wRw' and $w' \Vdash \mathsf{validmem}(m)$. We have $w \Vdash v : \mathsf{slot}(\mathtt{pc}, \Delta_p)$, and thus $w \Vdash v : \mathsf{slot}(\mathtt{pc}, \mathsf{codeptr}(\mathsf{top}))$. As the program is loaded at l, we also have $w' \Vdash v : \mathsf{slot}(\mathtt{pc}, \mathsf{just}\, l)$. By equation 72 we get $w' \Vdash v : \mathsf{safe}$, so by equation 66 we have $w' \Vdash v : \mathsf{safemem}(m)$. By equation 65, this means $\mathrm{safe}_k(v,m)$. □

We have established that our syntactic system is sound in the sense that every well typed program is safe. This follows from the fact that each individual typing rule is semantically valid in our model (from the definition of $\vdash$, equation 16). Even after we have proved Theorem 12.2 we may add more typing rules, as long as we derive them semantically.

13. Related work

13.1 Impredicative polymorphism with references

Here's a type isomorphism that one would expect to hold, in a system with continuation types and polymorphism:

$$\mathsf{codeptr}(\exists\alpha.\, F(\alpha)) \cong \forall\alpha.\, \mathsf{codeptr}(F(\alpha)). \quad (86)$$

Let's consider in particular a special case of the above,

$$\mathsf{codeptr}(\exists\alpha{:}\mathrm{Type}.\alpha \times A \times \mathsf{codeptr}(B)) \cong \forall\alpha{:}\mathrm{Type}.\mathsf{codeptr}(\alpha \times A \times \mathsf{codeptr}(B)) \quad (87)$$

where the left and right-hand sides are (de Morgan) dual ways of writing the CPS- and closure-converted type of a function $f : A \to B$. We're happy to say that the isomorphisms 86 and 87 hold in our system. We're also happy to say that in our system, none of the operations on values of quantified type—existential pack and unpack, type abstraction and application—require taking a step in the operational semantics. (As types are not manifest at runtime in a statically checked system, such steps would be effective no-ops, and therefore a nuisance.) To have both of these properties simultaneously is a technical advance over previous work [Ahm04], which presented two alternative models of quantifiers, each of which possessed one property but not the other.

We believe Ahmed's set-theoretic model can be adjusted (by simplifying the model of quantified types) to eliminate this nuisance. However, it is not obvious how to directly formalize such a repaired model—or equally, the mathematics of Section 6—in a logic without dependent types, in particular Church's higher-order logic (HOL). To our knowledge, no previous attempt to express equation 28 in HOL could do better than the solutionless 25. To avoid circularity in the HOL representation, Ahmed chooses the range of (HOL-encoded) Ψ to be a set of reified syntactic type expressions. This causes the failure of equation 87—in particular, the left side $(\mathsf{codeptr}(\exists\alpha.\ldots))$ has no useful semantics, while the right side $(\forall\alpha.\mathsf{codeptr}(\ldots))$ has the expected semantics. The Princeton FPCC compiler [CWAF03] works around this problem by carefully using only the $\forall$codeptr formulation of function-closures.

13.2 Approximation modality

The approximation modality is due to Nakano, who developed the idea in two papers. The first paper [Nak00] introduces a type system with recursive types and an approximation modality $\bullet$ (to which our $\triangleright$ is directly analogous), which for example allows the derivation

$$\vdash \mathbf{Y} : (\bullet X \to X) \to X,$$

where $\mathbf{Y}$ is the fixed-point combinator. The approximation modality allows, even in the presence of recursive types, a strongly normalizing calculus that may be sensibly used as a logic. The second paper [Nak01] generalizes the first, and makes precise the relationship to the intuitionistic version of the Gödel-Löb logic of provability.

Our work builds on and departs from Nakano's in two principal ways. First, we use the approximation modality in calculi that are by design capable of nontermination. Second, our main contribution is to show how the approximation modality is not just a modality for recursion, but also a modality for mutable references, and in general a modality for realizing inductive properties of small-step programs.

13.3 XCAP

Shao et al.'s Certified Assembly Programming [YHS03, NS06] is based directly on the calculus of inductive constructions. Like our system, CAP/XCAP supports separate verification of code modules, permits impredicative quantification and (with extensions) mutable references, and is expressive enough to specify invariants for assembly code. They achieve impredicativity by means of a syntactic specification of validity rules for impredicative propositions. This method does not permit elimination rules for the quantifiers, but they show how to work around the lack of elimination rules when existentials appear at top level in Hoare rules, which is good enough for a TAL.

14. Conclusion

Previous work broke functions and data structures into simpler constructs—that is, existential quantifiers, address arithmetic, unions, intersections, all applied to first-order continuations and immutable or mutable references. But here we have shown (equations 32, 62, 72) that first-order continuations (codeptr) and references (iref, ref) can themselves be broken down into simpler concepts, each of which is a *type* operator: safe models program safety, $\triangleright$ models induction over future steps, $\Box$ models type preservation, ! constructs logical statements about the current world. This decomposition allows simpler and finer reasoning about the interaction of instructions with types, and allows the easier construction of new types.

Our semantic model is powerful: it is the first model of references with impredicativity in which all the expected identities hold. It is versatile: it can express in a natural way all the types needed for typed assembly languages. It is modular: the vast majority of our definitions and lemmas apply equally to λ-calculus, von Neumann machines, and other useful abstract machines. It is compositional: there are a few simple primitives, from which everything else is constructed. It is sound: we have machine-checked proofs

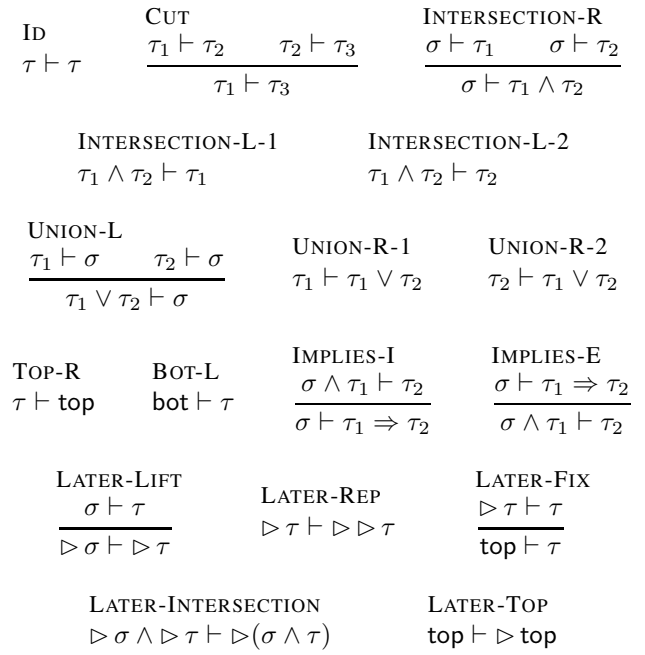

Figure 4. Gödel-Löb logic

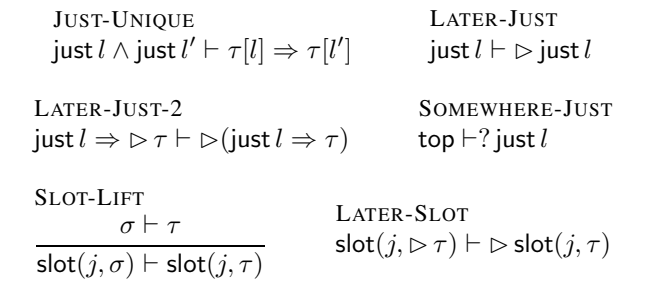

Figure 5. Just and slot

for the von Neumann version. It is operational: the statement of the soundness theorem guarantees directly an untyped notion of safety. Finally, it is *semantic:* the modal operator $\triangleright$ "later" explains the essence of approximation over the remaining steps of computation, in a way that allows us to better understand old types and better invent new ones.

Future work. Using this concise and easily representable model, we expect that machine-checked soundness proofs of usable TALs will be substantially smaller than those attempted to date [Cra03, App01].

Acknowledgments

We thank James Leifer, Amal Ahmed, and Zhong Shao for comments on earlier drafts of the paper.

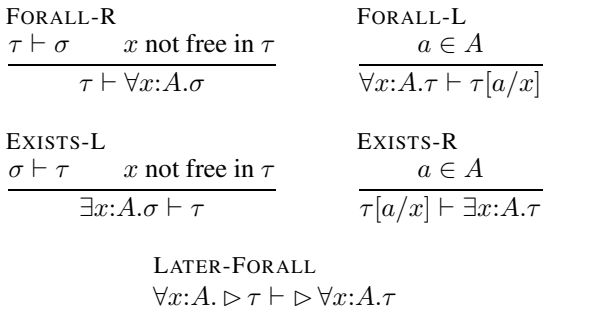

Figure 6. Quantification

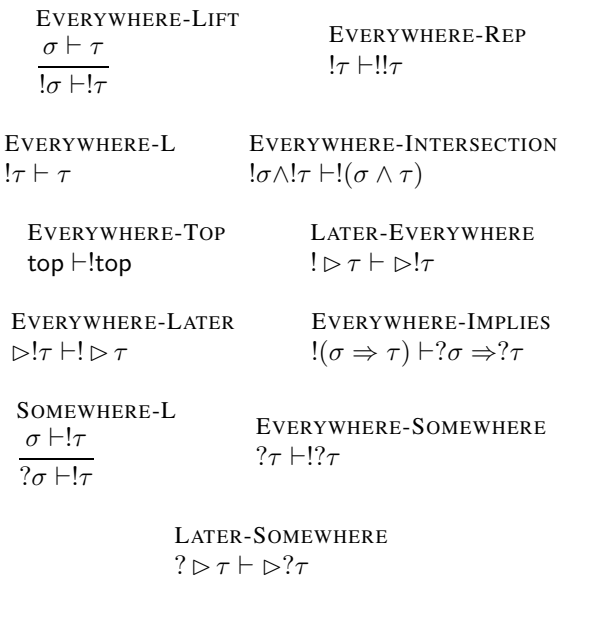

Figure 7. Quantification over values in a world

References

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *17th Annual IEEE Symp. on Logic in Computer Science*, pp. 75–86, June 2002.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: 27th* ACM SIGPLAN-SIGACT *Symp. on Principles of Programming Languages*, pp. 243–253, Jan. 2000.

[Ahm04] Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Princeton, NJ, Nov. 2004. Tech Report TR-713-04.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Symp. on Logic in Computer Science (LICS '01)*, pp. 247–258. IEEE, 2001.

[B+98] Bruno Barras et al. The Coq Proof Assistant reference manual. Technical report, INRIA, 1998.

[BW83] Michael Barr and Charles Wells. *Toposes, Triples and Theories*. Grundlehren der math. Wissenschaften. Springer Verlag, 1983. Reprint 12 in Theory and Applications of Category, //www.emis.de/journals/TAC/.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *POPL '03: 30th ACM Symp. on Principles of Programming Languages*, pp. 198–212, 2003.

[CWAF03] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *PLDI '03: Proc. 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 208–219, June 2003.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. In *Proceedings of a Colloquium: Modal and Many Valued Logics*, vol. 16, pp. 83–94, 1963.

[MM91] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 1991.

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

[MV05] Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *IEEE Symp. on Logic in Computer Science (LICS '05)*, 2005.

[Nak00] Hiroshi Nakano. A modality for recursion. In *LICS '00: 15th Annual* IEEE *Symp. on Logic in Computer Science*, pp. 255–266. IEEE Computer Society Press, 2000.

[Nak01] Hiroshi Nakano. Fixed-point logic with the approximation modality and its Kripke completeness. In *Theoretical Aspects of Computer Software*, vol. 2215 of *LNCS*, pp. 165–182. Springer, 2001.

[NS06] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *33rd ACM Symp. on Principles of Programming Languages*, pp. 320–333. ACM Press, Jan. 2006.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In M. Bezem and J. F. Groote, editors, *Proc. International Conference on Typed Lambda Calculi and Applications*, vol. 664, pp. 328–345. Springer Verlag Lecture Notes in Computer Science, 1993.

[TA06] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In *7th International Conference on Verifi cation, Model Checking and Abstract Interpretation (VMCAI'06)*, pp. 80–94, Jan. 2006.

[Tan05] Gang Tan. *A Compositional Logic for Control Flow and its Application to Foundational Proof-Carrying Code*. PhD thesis, Princeton University, Princeton, NJ, Aug. 2005. Tech Report CS-TR-731-05.

[Wu05] Dinghao Wu. *Interfacing Compilers, Proof Checkers, and Proofs for Foundational Proof-Carrying Code*. PhD thesis, Princeton University, Aug. 2005.

[YHS03] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symp. on Programming (ESOP'03)*, April 2003.

A. Allocation and initialization

Our type system can express the invariants of data structures that manage a region calculus (a typed malloc/free system, or a garbage collector). Such type systems are not new [AF00, CWAF03, Ahm04, Wu05]. Here we briefly illustrate how our new type system can express the necessary invariants just using the primitives we have already defined.

The simplest possible such data structure is an integer variable that defines the boundary between allocated and unallocated memory. We define $\mathsf{boundary}(l)$ to mean that every address $\geq l$ is unallocated, i.e. not in the domain of Ψ:

$$\mathsf{boundary}\ l \stackrel{\text{syn}}{=} \forall l'{:}\mathrm{Loc}.\forall \tau{:}\mathrm{Type}.\, l' \mapsto \tau \Rightarrow \mathsf{greater}(l, l') \tag{88}$$

Now we give a rule for type-checking $\textsc{store}(i, 0, j)$ (i.e., $m(r(i) + 0) \leftarrow r(j)$) when it serves to initialize a new reference:

$$\frac{\begin{array}{c} n \text{ encodes } \textsc{store}(i,0,j) \\ \tau \text{ necessary} \qquad \phi \text{ necessary} \qquad \mathrm{notindom}(\mathsf{pc}, \phi) \\ P = \phi \wedge \exists l{:}\mathrm{Loc}.\mathsf{boundary}\, l \wedge \mathsf{slot}(i, \mathsf{just}\, l) \wedge \mathsf{slot}(j, \tau) \\ Q = \phi \wedge \exists l{:}\mathrm{Loc}.\mathsf{boundary}(l+1) \wedge \mathsf{slot}(i, \mathsf{just}\, l \wedge \mathsf{ref}\, \tau) \end{array}}{\mathsf{ref}(\mathsf{just}\, n),\ \triangleright\, \mathsf{offset}(1, \mathsf{codeptr}(Q)) \vdash \mathsf{codeptr}(P)} \tag{89}$$

Before the instruction, register i points at the boundary. The instruction does not modify i, but does modify the boundary, in that the domain of Ψ in the new world has moved.

The same store instruction, when it is used for the different purpose of updating an existing ref, has this typing rule:

$$\frac{\begin{array}{c} n \text{ encodes } \textsc{store}(i,0,j) \qquad B = \mathsf{boundary}\, l \\ \tau \text{ necessary} \qquad \phi \text{ necessary} \qquad \mathrm{notindom}(\mathsf{pc}, \phi) \\ P = B \wedge \phi \wedge \mathsf{slot}(i, \mathsf{ref}\, \tau) \wedge \mathsf{slot}(j, \tau) \end{array}}{\mathsf{ref}(\mathsf{just}\, n),\ \triangleright\, \mathsf{offset}(1, \mathsf{codeptr}(B \wedge \phi)) \vdash \mathsf{codeptr}(P)} \tag{90}$$

Of course, both of these rules for STORE must be proved as derived lemmas from the definition in our model. By proving every typing rule as a derived lemma, we thus prove soundness of the rules.