A Kripke Logical Relation Between ML and Assembly

Chung-Kil Hur * Derek Dreyer
Max Planck Institute for Software Systems (MPI-SWS)
{gil,dreyer}@mpi-sws.org

Abstract

There has recently been great progress in proving the correctness of compilers for increasingly realistic languages with increasingly realistic runtime systems. Most work on this problem has focused on proving the correctness of a particular compiler, leaving open the question of how to verify the correctness of assembly code that is hand-optimized or linked together from the output of multiple compilers. This has led Benton and other researchers to propose more abstract, compositional notions of when a low-level program correctly realizes a high-level one. However, the state of the art in so-called "compositional compiler correctness" has only considered relatively simple high-level and low-level languages.

In this paper, we propose a novel, extensional, compiler-independent notion of equivalence between high-level programs in an expressive, impure ML-like λ-calculus and low-level programs in an (only slightly) idealized assembly language. We define this equivalence by means of a biorthogonal, step-indexed, Kripke logical relation, which enables us to reason quite flexibly about assembly code that uses local state in a different manner than the high-level code it implements (*e.g.,* self-modifying code). In contrast to prior work, we factor our relation in a symmetric, language-generic fashion, which helps to simplify and clarify the formal presentation, and we also show how to account for the presence of a garbage collector. Our approach relies on recent developments in Kripke logical relations for ML-like languages, in particular the idea of possible worlds as state transition systems.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Step-indexed Kripke logical relations, biorthogonality, compositional compiler correctness, garbage collection, self-modifying code

* This work was undertaken while the first author was at PPS, Université Paris Diderot, supported by Digiteo/Ile-de-France project COLLODI (2009-28HD) and Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) Grant R11-2008-007-01002-0.

1. Introduction

While compiler verification is an age-old problem, there has been remarkable progress in the last several years in proving the correctness of compilers for increasingly realistic languages with increasingly realistic runtime systems. Of particular note is Leroy's Compcert project [18], in which he used the Coq proof assistant to both program and verify a multi-pass optimizing compiler from Cminor (a C-like intermediate language) to PowerPC assembly. Dargaye [13] has adapted the Compcert framework to a compiler for a pure mini-ML language, and McCreight *et al.* [19] have extended it to support interfacing with a garbage collector. Independently, Chlipala [10, 12] has developed verified compilers for both pure and impure functional core languages, the former garbage-collected, with a focus on using custom Coq tactics to provide significant automation of verification.

That said, all of the aforementioned work has focused on proving the correctness of a *particular* compiler, leaving open the question of how to verify the correctness of assembly code that is hand-optimized or linked together from the output of multiple compilers. The issue is that compiler correctness results are typically established by exhibiting a fairly close simulation relation between source and target code, but code produced by another compiler may obey an entirely different simulation relation with the source program, and hand-optimized code might not closely simulate the source program at all. Thus, existing correctness proofs depend fundamentally on the "closed-world" assumption that one has control over how the whole source program is compiled.

In order to lift the closed-world assumption, Benton and Hur [5] suggest that what is needed is a more abstract, extensional notion of what it means for a low-level program to correctly implement a high-level one—a notion that is not tied to a particular compiler and that, moreover, offers as much flexibility in the low-level representation of high-level features as possible. When reasoning strictly about high-level programs, the canonical extensional notion of when one program implements the same functionality as another is *observational* (or *contextual*) equivalence, which says that the two programs exhibit the same (termination) behavior when placed into the context of an arbitrary enclosing well-typed high-level program. However, it is not clear how to define such a contextual notion of equivalence between high- and low-level programs, because there is no way to run both programs under the *same* context—one would need to quantify over *equivalent* high- and low-level program contexts, but when are two contexts equivalent? We are back to the original question.

Benton and Hur's solution is to define a *logical relation* between the high- and low-level languages (actually two relations, one for each direction of semantic approximation, employing a denotational semantics to represent the high-level side). Logical relations are inherently extensional—*e.g.,* two functions are logically related iff they map related arguments to related results, regardless of their private implementation details—and guarantee equivalent termination behavior under arbitrary contexts that are themselves logically related. While not as canonical as contextual equivalence, logical equivalence is nevertheless useful as long as one can establish that the logical relations are sufficiently populated—*i.e.,* that they relate enough programs/contexts of interest.

In the traditional setting where one is defining equivalence of programs in the same language, this "sufficient population" property is ensured by the *fundamental theorem of logical relations*, which states that all well-typed programs (and thus all well-typed contexts) are logically self-related. For mixed high-low relations, Benton and Hur demonstrate sufficient population by providing a simple, one-pass compilation translation from their high- to their low-level language and proving that all well-typed high-level programs are logically equivalent to their compilations. They also use the logical relations to show the relatedness of some simple hand-optimized low-level code with corresponding high-level programs.

Benton and Hur present their work as the first step towards "compositional compiler correctness". However, the source language they consider—the simply-typed λ-calculus with recursion—is purely functional, and the target language they consider—an SECD machine—is relatively high-level. Chlipala [11] has subsequently proposed a more syntactic approach to proving compositional compiler correctness, applicable to a richer, impure (albeit untyped) source language, but the target language he considers is also pretty high-level, namely a CPS variant of the source.

1.1 Contributions

In this paper, we study compositional equivalence of high- and low-level programs in a more realistic setting. Our high-level language is an expressive ML-like CBV λ-calculus, supporting abstract types, general recursive types, and general mutable references. Our low-level language is an (only slightly) idealized assembly language. Furthermore, our logical relation is designed to be sound in the presence of garbage collection, under some fairly abstract assumptions about the behavior of the garbage collector that are satisfied by both mark-and-sweep and copying collectors.

Following Benton and Hur, we define our equivalence using a *biorthogonal, step-indexed* logical relation. Biorthogonality is useful when reasoning about programs (such as low-level ones) whose behavior is *context-sensitive*, and step-indexing is useful in reasoning about semantically "cyclic" features like recursive types and higher-order state.

We depart from prior work, though, in that our relation is also a *Kripke* logical relation—*i.e.,* it is indexed by *possible worlds* that specify assumptions about the machine state. Possible worlds are useful in enforcing invariants about low-level data structures (*e.g.,* that a heap-allocated representation of a closure is immutable). They are also helpful in encoding a variety of runtime system invariants, such as the convention concerning callee-save registers and the notion of data liveness. Last but not least, possible worlds enable us to reason quite flexibly about assembly code that uses *local* state in a different manner than the high-level code it implements. An interesting example of this is *self-modifying* assembly code, whose correctness proof involves reasoning about low-level internal state changes—specifically, changes to the code itself—that clearly have no high-level counterpart. This is the essence of what we mean when we say that our relation is *extensional.*

Technically, our approach relies closely on recent developments in Kripke logical relations for ML-like languages, in particular the idea of possible worlds as *state transition systems*. This idea, which we review in Section 3.3, was proposed originally by Ahmed *et al.* [2] (in a somewhat different form) as a way to reason about representation independence for so-called "generative" abstract data types, whose private state undergoes a controlled series of state transitions during the execution of the program. Dreyer *et al.* [14] have subsequently generalized the idea in order to reason about "well-bracketed" state changes. By basing our high-low logical relations on this most recent work, we are able to cleanly model a variety of state transition systems that arise naturally in low-level code (*e.g.,* in self-modifying code).

Lastly, a novel feature of our logical relation is that, while it is defined by induction on an ML-like type structure, it is also defined in a *language-generic* fashion. That is, it may be instantiated to form an equivalence relation between any two languages (high- or low-level) that are capable of implementing various relevant linguistic constructs—*e.g.,* function application, plugging a continuation with a value, etc. Factoring the relation in this way helps to simplify and clarify the formal presentation. Moreover, it has the advantage that the relation becomes inherently symmetric and thus easier to use in proving high-low equivalences than Benton and Hur's asymmetric approximation relations.

2. HIGH and LOW

The high-level language **HIGH** is a System F-like polymorphic λ-calculus extended with existential, product and iso-recursive types, as well as ML-style general references (higher-order state). Figure 1 shows the syntax and the typing and evaluation judgments. The inference rules for typing and evaluation are standard, so we omit them (see the companion technical appendix [15] for details).

The low-level language **LOW** is an assembly language idealized in two ways: its word and memory sizes are infinite and its instructions are represented by abstract objects rather than by physical words. Its memory consists of four entities: code memory, register file, stack and heap. A code memory is a map from physical addresses (represented by natural numbers) to instructions. A register file is a map from registers to words. Words in turn are represented by natural numbers with an extra bit indicating whether the word is a pointer to a heap cell or not (useful for garbage collection purposes). There are 12 registers, half of which (sp,$\mathrm{sv}_0, \ldots, \mathrm{sv}_4$) are specified as callee-save registers by our calling convention. Both a stack and a heap are random-access memories (*i.e.,* maps from addresses to words).

The instruction set includes standard instructions (`jmp`, `jnz`, `jneq`, `move`, `plus`, `minus`) supporting different addressing modes via lvalues and rvalues. The non-standard operations include `halt` for normal termination, `fail` for raising a runtime error, `jptr` for testing whether a value is a pointer or not, `setptr` for marking a pointer bit, and `isr`, `isw` for inspecting and updating instructions in code memory. Note that as instructions are not represented by words, we employ a bijection $\mathbb{E}$ and its inverse $\mathbb{D}$ (for $\mathbb{E}$ncode and $\mathbb{D}$ecode) to convert back and forth between instructions and words.

The dynamic semantics of **LOW** is standard and is given in Figure 1. A machine configuration (Φ, pc) is a pair consisting of a memory and a program counter; it halts, fails or evolves to another configuration (Φ', pc') by executing the instruction stored at pc in the code memory $\Phi.\mathrm{code}$.

3. The Key Ideas

In this section, we present the key ideas behind our work through the lens of an illustrative, challenging example. We will walk through the code of this example, suggest intuitively how to reason about it, and then explain how our Kripke logical relation formalizes this intuition. While we will initially ignore the question of how garbage collection affects matters, we will return to this issue at the end of the section and discuss how our logical relation enables reasoning in the presence of a garbage collector.

This section is intended to be accessible to a broad programming-languages audience, and hopefully to serve as a useful guide to the

HIGH – Syntax & Semantics

$$\begin{array}{lll}
\tau & ::= & \alpha \mid b \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mu\alpha.\,\tau \mid \mathsf{ref}\ \tau \\
e & ::= & x \mid \ell \mid \langle e_1, e_2\rangle \mid e.1 \mid e.2 \mid \lambda x{:}\tau.\,e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau \mid \\
 & & \mathsf{pack}\ \langle\tau_1, e\rangle\ \mathsf{as}\ \tau_2 \mid \mathsf{unpack}\ e_1\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ e_2 \mid \\
 & & \mathsf{roll}_\tau\ e \mid \mathsf{unroll}\ e \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid !e \mid e_1 == e_2 \mid \ldots \\
v & ::= & x \mid \ell \mid \langle v_1, v_2\rangle \mid \lambda x{:}\tau.\,e \mid \Lambda\alpha.e \mid \mathsf{pack}\ \langle\tau_1, v\rangle\ \mathsf{as}\ \tau_2 \mid \mathsf{roll}_\tau\ v \mid \ldots \\
K & ::= & \bullet \mid \langle K, e_2\rangle \mid \langle v_1, K\rangle \mid K.1 \mid K.2 \mid K\ e_2 \mid v_1\ K \mid K\ \tau \mid \mathsf{roll}_\tau\ K \mid \\
 & & \mathsf{unroll}\ K \mid \mathsf{pack}\ \langle\tau_1, K\rangle\ \mathsf{as}\ \tau_2 \mid \mathsf{unpack}\ K\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ e_2 \mid \\
 & & \mathsf{ref}\ K \mid K := e_2 \mid v_1 := K \mid !K \mid K == e_2 \mid v_1 == K \mid \ldots
\end{array}$$

$\Sigma ::= \cdot \mid \Sigma, \ell{:}\tau$ with $\mathrm{ftv}(\tau) = \emptyset$ $\quad \Delta ::= \cdot \mid \Delta, \alpha \quad \Gamma ::= \cdot \mid \Gamma, x{:}\tau$

Static semantics : $\Sigma; \Delta; \Gamma \vdash e : \tau$

$\mathrm{HCVal} \stackrel{\text{def}}{=} \{\, v \mid \mathrm{ftv}(v) = \emptyset \wedge \mathrm{fv}(v) = \emptyset \,\}$

$\mathrm{HHeap} \stackrel{\text{def}}{=} \{\, h \in \mathrm{HLoc} \rightharpoonup_{\text{fin}} \mathrm{HCVal} \,\}$

Dynamic semantics : $(h, e) \hookrightarrow (h', e')$

LOW – Syntax

$$\begin{array}{l}
\mathrm{PConf} \stackrel{\text{def}}{=} \{\, (\Phi, \mathrm{pc}) \in \mathrm{PMem} \times \mathrm{PAddr} \,\} \\
\mathrm{PMem} \stackrel{\text{def}}{=} \{\, \Phi = (\mathrm{code}, \mathrm{reg}, \mathrm{stk}, \mathrm{hp}) \\
\qquad\qquad \in \mathrm{PCode} \times \mathrm{PRegFile} \times \mathrm{PStack} \times \mathrm{PHeap} \,\}
\end{array}$$

$$\begin{array}{llll}
\mathrm{PCode} \stackrel{\text{def}}{=} \mathrm{PAddr} \to \mathrm{Instruction} & & \mathrm{PRegFile} \stackrel{\text{def}}{=} \mathrm{Register} \to \mathrm{PWord} \\
\mathrm{PStack} \stackrel{\text{def}}{=} \mathrm{PAddr} \to \mathrm{PWord} & & \mathrm{PHeap} \stackrel{\text{def}}{=} \mathrm{PAddr} \to \mathrm{PWord} \\
\mathrm{PAddr} \stackrel{\text{def}}{=} \{\, a \in \mathbb{N} \,\} & & \mathrm{PWord} \stackrel{\text{def}}{=} \{\, w \in \{\,0, 1\,\} \times \mathbb{N} \,\}
\end{array}$$

$$\begin{array}{lll}
r \in \mathrm{Register} & ::= & \mathrm{sp} \mid \mathrm{sv}_0 \mid \ldots \mid \mathrm{sv}_4 \mid \mathrm{wk}_0 \mid \ldots \mid \mathrm{wk}_5 \\
\mathrm{lv} \in \mathrm{PLvalue} & ::= & \lfloor r \rfloor \mid \langle a\rangle_{\mathrm{s}} \mid \langle r - o\rangle_{\mathrm{s}} \mid \langle a\rangle_{\mathrm{h}} \mid \langle r + o\rangle_{\mathrm{h}} \\
\mathrm{rv} \in \mathrm{PRvalue} & ::= & \mathrm{lv} \mid w \\
\iota \in \mathrm{Instruction} & ::= & \mathtt{fail} \mid \mathtt{halt} \mid \mathtt{jmp}\ \mathrm{rv} \mid \mathtt{jnz}\ \mathrm{rv}\ \mathrm{rv} \mid \mathtt{jneq}\ \mathrm{rv}\ \mathrm{rv}\ \mathrm{rv} \mid \\
 & & \mathtt{jptr}\ \mathrm{rv}\ \mathrm{rv} \mid \mathtt{setptr}\ \mathrm{lv} \mid \mathtt{move}\ \mathrm{lv}\ \mathrm{rv} \mid \mathtt{plus}\ \mathrm{lv}\ \mathrm{rv}\ \mathrm{rv} \mid \\
 & & \mathtt{minus}\ \mathrm{lv}\ \mathrm{rv}\ \mathrm{rv} \mid \mathtt{isr}\ \mathrm{lv}\ \mathrm{rv} \mid \mathtt{isw}\ \mathrm{rv}\ \mathrm{rv}
\end{array}$$

LOW – Semantics

$$|w| \stackrel{\text{def}}{=} \pi_2(w) \quad \mathrm{isptr}(w) \stackrel{\text{def}}{=} (\pi_1(w) = 1) \quad \underline{n} \stackrel{\text{def}}{=} (0, n) \quad \widehat{a} \stackrel{\text{def}}{=} (1, a)$$

$$\begin{array}{llll}
\Phi(w) & \stackrel{\text{def}}{=} w & \Phi(\lfloor r\rfloor) & \stackrel{\text{def}}{=} \Phi.\mathrm{reg}(r) \\
\Phi(\langle a\rangle_{\mathrm{s}}) & \stackrel{\text{def}}{=} \Phi.\mathrm{stk}(a) & \Phi(\langle r - o\rangle_{\mathrm{s}}) & \stackrel{\text{def}}{=} \Phi.\mathrm{stk}(|\Phi(\lfloor r\rfloor)| - o) \\
\Phi(\langle a\rangle_{\mathrm{h}}) & \stackrel{\text{def}}{=} \Phi.\mathrm{hp}(a) & \Phi(\langle r + o\rangle_{\mathrm{h}}) & \stackrel{\text{def}}{=} \Phi.\mathrm{hp}(|\Phi(\lfloor r\rfloor)| + o)
\end{array}$$

$$\begin{array}{lll}
\Phi[\lfloor r\rfloor \mapsto w] & \stackrel{\text{def}}{=} & (\Phi.\mathrm{code}, \Phi.\mathrm{reg}[r \mapsto w], \Phi.\mathrm{stk}, \Phi.\mathrm{hp}) \\
\Phi[\langle a\rangle_{\mathrm{s}} \mapsto w] & \stackrel{\text{def}}{=} & (\Phi.\mathrm{code}, \Phi.\mathrm{reg}, \Phi.\mathrm{stk}[a \mapsto w], \Phi.\mathrm{hp}) \\
\Phi[\langle r - o\rangle_{\mathrm{s}} \mapsto w] & \stackrel{\text{def}}{=} & (\Phi.\mathrm{code}, \Phi.\mathrm{reg}, \Phi.\mathrm{stk}[|\Phi(\lfloor r\rfloor)| - o \mapsto w], \Phi.\mathrm{hp}) \\
\Phi[\langle a\rangle_{\mathrm{h}} \mapsto w] & \stackrel{\text{def}}{=} & (\Phi.\mathrm{code}, \Phi.\mathrm{reg}, \Phi.\mathrm{stk}, \Phi.\mathrm{hp}[a \mapsto w]) \\
\Phi[\langle r + o\rangle_{\mathrm{h}} \mapsto w] & \stackrel{\text{def}}{=} & (\Phi.\mathrm{code}, \Phi.\mathrm{reg}, \Phi.\mathrm{stk}, \Phi.\mathrm{hp}[|\Phi(\lfloor r\rfloor)| + o \mapsto w])
\end{array}$$

$$\begin{array}{lll}
[\![\mathtt{fail}]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & \mathit{fail} \\
[\![\mathtt{halt}]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & \mathit{halt} \\
[\![\mathtt{jmp}\ \mathrm{rv}]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & (\Phi, |\Phi(\mathrm{rv})|) \\
[\![\mathtt{jnz}\ \mathrm{rv}_1\ \mathrm{rv}_2]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & \text{if } \Phi(\mathrm{rv}_2) \neq \underline{0} \\
 & & \text{then } (\Phi, |\Phi(\mathrm{rv}_1)|) \text{ else } (\Phi, \mathrm{pc} + 1) \\
[\![\mathtt{jneq}\ \mathrm{rv}_1\ \mathrm{rv}_2\ \mathrm{rv}_3]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & \text{if } \Phi(\mathrm{rv}_2) \neq \Phi(\mathrm{rv}_3) \\
 & & \text{then } (\Phi, |\Phi(\mathrm{rv}_1)|) \text{ else } (\Phi, \mathrm{pc} + 1) \\
[\![\mathtt{jptr}\ \mathrm{rv}_1\ \mathrm{rv}_2]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & \text{if } \mathrm{isptr}(\Phi(\mathrm{rv}_2)) \\
 & & \text{then } (\Phi, |\Phi(\mathrm{rv}_1)|) \text{ else } (\Phi, \mathrm{pc} + 1) \\
[\![\mathtt{move}\ \mathrm{lv}\ \mathrm{rv}]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & (\Phi[\mathrm{lv} \mapsto \Phi(\mathrm{rv})], \mathrm{pc} + 1) \\
[\![\mathtt{setptr}\ \mathrm{lv}]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & (\Phi[\mathrm{lv} \mapsto \widehat{|\Phi(\mathrm{lv})|}], \mathrm{pc} + 1) \\
[\![\mathtt{plus}\ \mathrm{lv}\ \mathrm{rv}_1\ \mathrm{rv}_2]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & (\Phi[\mathrm{lv} \mapsto \underline{|\Phi(\mathrm{rv}_1)| + |\Phi(\mathrm{rv}_2)|}], \mathrm{pc} + 1) \\
[\![\mathtt{minus}\ \mathrm{lv}\ \mathrm{rv}_1\ \mathrm{rv}_2]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & (\Phi[\mathrm{lv} \mapsto \underline{|\Phi(\mathrm{rv}_1)| - |\Phi(\mathrm{rv}_2)|}], \mathrm{pc} + 1) \\
[\![\mathtt{isr}\ \mathrm{lv}\ \mathrm{rv}]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & (\Phi[\mathrm{lv} \mapsto \underline{\mathbb{E}(\Phi.\mathrm{code}(|\Phi(\mathrm{rv})|))}], \mathrm{pc} + 1) \\
[\![\mathtt{isw}\ \mathrm{rv}_1\ \mathrm{rv}_2]\!]\,(\Phi, \mathrm{pc}) & \stackrel{\text{def}}{=} & ((\Phi.\mathrm{code}[\,|\Phi(\mathrm{rv}_1)| \mapsto \mathbb{D}(|\Phi(\mathrm{rv}_2)|)], \\
 & & \quad \Phi.\mathrm{reg}, \Phi.\mathrm{stk}, \Phi.\mathrm{hp}), \mathrm{pc} + 1)
\end{array}$$

where $\mathbb{E}$: Instruction $\to \mathbb{N}$ is a bijection, and $\mathbb{D} = \mathbb{E}^{-1}$.

Dynamic semantics : $(\Phi, \mathrm{pc}) \hookrightarrow [\![\Phi.\mathrm{code}(\mathrm{pc})]\!]\,(\Phi, \mathrm{pc})$

Figure 1. Syntax and Semantics for the **HIGH** and **LOW** Languages (excerpt)

vast majority of readers who are not intimately familiar with recent developments in Kripke logical relations.

3.1 A Motivating Example

Our motivating example is based on Pitts and Stark's "awkward" example [22]. Their original example is almost blindingly simple—prove that the following **HIGH** terms are contextually equivalent:

$$\begin{array}{lll}
e & \stackrel{\text{def}}{=} & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f{:}\mathsf{unit} \to \mathsf{unit}.\,(x := 1; f\ \langle\rangle; !x) \\
e' & \stackrel{\text{def}}{=} & \lambda f{:}\mathsf{unit} \to \mathsf{unit}.\,(f\ \langle\rangle; 1)
\end{array}$$

The first term, e, allocates a fresh memory location x, initially set to 0, and then returns a higher-order function. When called, the latter will set x to 1, invoke its callback argument f, and then return the contents of x. The second term, e', is similar, except that it does not bother allocating or updating any memory, and the function it defines always returns 1. Proving that e and e' are contextually equivalent is tantamount to showing that, in the former, whenever the callback invocation $f\ \langle\rangle$ returns, x points to 1.

After a moment's thought, it should hopefully be intuitively clear why the equivalence holds. The pointer x in e is initially set to 0, but once the function that e returns is applied for the first time, x will be set to 1 and will never be set back to 0. This is because e "owns" x as a piece of *local state* and the only thing e ever does to x after first allocating it is to set it to 1. Thus, the "awkward" example serves as an elegant distillation of (1) the ability of code to control some local state and impose arbitrary constraints on it, and (2) the ability of that local state to evolve over time in an *irreversible* (or *monotone*) way. Such irreversible changes to local state arise in a variety of real-world situations—*e.g.,* in "generative" ADTs (whose sets of inhabitants grow over time), and in data structure initialization [2]. It is thus rather remarkable that only recently have methods been developed for proving an equivalence as simple as the "awkward" example [8, 2, 23]. (More on that in Section 3.3.)

We are now ready to present our motivating example. We want to prove a variant of the awkward example, namely that the **HIGH** term e is implemented correctly by a **LOW** program p, where p's implementation follows the second **HIGH** term e' fairly closely. By itself, that would already be an interesting result—it would demonstrate the extensional equivalence of a high-level program with an "optimized" low-level program, where the optimization is based on the inability of high-level program contexts to observe e's manipulation of local state. But we will make it more interesting still with an added twist: the code of the function that p evaluates to will be *obfuscated* using a primitive form of encryption, and when first applied, the function will first decrypt and overwrite itself via *self-modifying code*.

The **LOW** program p is shown in Figure 2. Before we walk through the code, let us first note that p is parameterized by alloc, a code pointer to the memory allocation routine, and bg, the location in the code segment where p's code will be loaded and where its execution will begin—these parameters will be instantiated as part of linking and loading. (We will define the formal semantics of linking and loading for **LOW** programs in Section 7.)

bg: Create and return a closure. The evaluation of p is very simple: it does no interesting computation except to immediately create and return a closure value, just as the **HIGH** term e' does. The first 3 instructions starting at bg allocate a fresh one-word closure on the heap by invoking the alloc routine (passing it the size parameter 1 in register wk_5 and the return address bg + 3 in register wk_4). We only need one word for the closure because the function we're implementing (e') is closed, so all we need to store in the closure is the naked code pointer. The alloc routine is assumed to return the pointer to a fresh one-word cell in wk_5, without modifying the contents of any registers except wk_4 and

$e \overset{\text{def}}{=} \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f{:}\mathsf{unit} \to \mathsf{unit}.\ x := 1; f\ \langle\rangle; !x$

$p \overset{\text{def}}{=} \lambda\, \mathrm{alloc}, \mathrm{bg}.\ [$

bg	`move`	$\lfloor \mathrm{wk}_4 \rfloor$	$\underline{\mathrm{bg}+3}$		
	`move`	$\lfloor \mathrm{wk}_5 \rfloor$	$\underline{1}$		
	`jmp`	$\underline{\mathrm{alloc}}$			
$\mathrm{bg}+3$	`move`	$\langle \mathrm{wk}_5 + 0\rangle_{\mathrm{h}}$	$\underline{\mathrm{bg}+5}$		
	`jmp`	$\lfloor \mathrm{wk}_0 \rfloor$			
$\mathrm{bg}+5$	`move`	$\lfloor \mathrm{wk}_3 \rfloor$	$\underline{\mathrm{bg}+10}$		
$\mathrm{bg}+6$	`isr`	$\lfloor \mathrm{wk}_4 \rfloor$	$\lfloor \mathrm{wk}_3 \rfloor$		
	`minus`	$\lfloor \mathrm{wk}_4 \rfloor$	$\lfloor \mathrm{wk}_4 \rfloor$	$\underline{666}$	
	`isw`	$\lfloor \mathrm{wk}_3 \rfloor$	$\lfloor \mathrm{wk}_4 \rfloor$		
	`plus`	$\lfloor \mathrm{wk}_3 \rfloor$	$\lfloor \mathrm{wk}_3 \rfloor$	$\underline{1}$	
$\mathrm{bg}+10$	$\mathbb{D}(\mathbb{E}($`jneq`	$\mathrm{bg}+6$	$\lfloor \mathrm{wk}_3 \rfloor$	$\underline{\mathrm{bg}+21}$	$)+666)$
$\mathrm{bg}+11$	$\mathbb{D}(\mathbb{E}($`isw`	$\mathrm{bg}+5$	$\underline{\mathbb{E}(\texttt{jmp}\ \mathrm{bg}+12)}$		$)+666)$
$\mathrm{bg}+12$	$\mathbb{D}(\mathbb{E}($`move`	$\langle \mathrm{wk}_1 + 0\rangle_{\mathrm{h}}$	$\underline{\mathrm{bg}+13}$		$)+666)$
$\mathrm{bg}+13$	$\mathbb{D}(\mathbb{E}($`plus`	$\lfloor \mathrm{sp} \rfloor$	$\lfloor \mathrm{sp} \rfloor$	$\underline{1}$	$)+666)$
	$\mathbb{D}(\mathbb{E}($`move`	$\langle \mathrm{sp} - 1\rangle_{\mathrm{s}}$	$\lfloor \mathrm{wk}_0 \rfloor$		$)+666)$
	$\mathbb{D}(\mathbb{E}($`move`	$\lfloor \mathrm{wk}_1 \rfloor$	$\lfloor \mathrm{wk}_2 \rfloor$		$)+666)$
	$\mathbb{D}(\mathbb{E}($`move`	$\lfloor \mathrm{wk}_0 \rfloor$	$\underline{\mathrm{bg}+18}$		$)+666)$
	$\mathbb{D}(\mathbb{E}($`jmp`	$\langle \mathrm{wk}_1 + 0\rangle_{\mathrm{h}}$			$)+666)$
$\mathrm{bg}+18$	$\mathbb{D}(\mathbb{E}($`move`	$\lfloor \mathrm{wk}_5 \rfloor$	$\underline{1}$		$)+666)$
	$\mathbb{D}(\mathbb{E}($`minus`	$\lfloor \mathrm{sp} \rfloor$	$\lfloor \mathrm{sp} \rfloor$	$\underline{1}$	$)+666)$
$\mathrm{bg}+20$	$\mathbb{D}(\mathbb{E}($`jmp`	$\langle \mathrm{sp} - 0\rangle_{\mathrm{s}}$			$)+666)$

$]$

Figure 2. Motivating Example

wk_5. We then store in that cell the code pointer $\mathrm{bg} + 5$, before jumping to the return address, which we assume the linker/loader had passed to p originally in register wk_0. (The linker/loader has essentially the same calling convention as for ordinary functions, which is different from the one for alloc and is described below—see $\mathrm{bg} + 12$.) Although our calling convention is that return values are passed back to the caller in wk_5, our return value was already in wk_5 after the call to alloc, so we need not explicitly move anything into wk_5 before returning.

We now describe the implementation of the closure returned (in wk_5) by the evaluation of p. Initially, this closure contains just a code pointer to $\mathrm{bg} + 5$, but eventually that code pointer will be updated (see below).

$\mathrm{bg} + 5$: Decrypt and overwrite the code. The code from $\mathrm{bg} + 10$ on is obfuscated by the addition of 666 to the machine representation of each instruction. Eventually, once the code is decrypted, the function will be executable starting at the address $\mathrm{bg} + 13$. Before that time, however, the function must begin executing at $\mathrm{bg} + 5$, because the first step will be to decrypt and overwrite the obfuscated instructions. The reader can easily verify manually that the code starting at $\mathrm{bg} + 5$ will use register wk_3 to loop through the instructions $\mathrm{bg} + 10$ through $\mathrm{bg} + 20$. For each, it will use `isr` to read the instruction stored at wk_3 into wk_4, subtract 666 from it, and then write the decrypted instruction back to the code segment at address wk_3. When this loop is finished, the program counter will be at $\mathrm{bg} + 11$.

$\mathrm{bg} + 11$: Redirect the first instruction of the function. Having decrypted the code, we do not want future calls to this function to perform the decryption again. We therefore overwrite the first instruction (at $\mathrm{bg} + 5$) with a jump to $\mathrm{bg} + 12$.

$\mathrm{bg} + 12$: Update the code pointer. The **LOW** calling convention is that a function is passed its return address in wk_0, its argument in wk_2, and its own closure in wk_1. The decryption code starting at $\mathrm{bg} + 5$ did not touch any of these registers, so at $\mathrm{bg} + 12$ we know that wk_1 still stores a closure with a code pointer to $\mathrm{bg} + 5$. Having decrypted the code, we can now safely update this code pointer—$\langle \mathrm{wk}_1 + 0\rangle_{\mathrm{h}}$—to point to $\mathrm{bg} + 13$, the address where the function begins its computation in earnest.

An important point: Why did we bother *both* redirecting the $\mathrm{bg} + 5$ instruction (to jump to $\mathrm{bg} + 12$) *and* updating the code pointer in the function closure? Would the latter alone not have been sufficient? The answer is that it depends. If p were only evaluated once, in which case only one closure for this function were ever generated, then yes, just updating the code pointer would be sufficient because $\mathrm{bg} + 5$ would become effectively dead code. But we would like our notion of high-low program equivalence to be preserved under a rich set of program contexts including those that evaluate the programs—here, p and e—more than once. Repeated evaluation of p will result in the creation of multiple closures for the function p defines, and merely updating the code pointer for one closure will not change the fact that other closures may still point to $\mathrm{bg} + 5$, so it is necessary to redirect the $\mathrm{bg} + 5$ instruction as well.

$\mathrm{bg} + 13$: Implement $\lambda f.\ (f\ \langle\rangle; 1)$. This is the implementation of the function proper. We first push our return address wk_0 onto the stack. We then invoke the callback argument (f in the **HIGH** code) by moving a pointer to f's closure into wk_1, moving the return address $\mathrm{bg} + 18$ into wk_0, and jumping to f's code pointer $\langle \mathrm{wk}_1 + 0\rangle_{\mathrm{h}}$. (Note: f's argument type is unit, so there is no need to pass anything in the argument register wk_2.) When control is returned to $\mathrm{bg} + 18$, we store the result 1 in the result register wk_5, pop the return address off the stack, and jump to it.

3.2 Discussion of the Motivating Example

Why does p implement e? Intuitively, the reason is that the self-modifying aspects of p are not visible to p's clients because they do not affect its extensional behavior; and ultimately, once p has decrypted itself, it behaves essentially the same as the **HIGH** term e', which we have already argued is equivalent to e. Of course, this begs the question: how exactly do we know that p's clients cannot observe its self-modifications?

Interestingly, the answer is remarkably similar to the argument for why e and e' are equivalent, namely that what p does to its own code takes the form of irreversible changes to local state. Specifically, we take it as a given that p "owns" its own code, and since the evaluation of p will allocate a fresh memory cell for the closure it returns, p owns that closure as well. Thus, in reasoning about p, we can place restrictions on how its code and closure may evolve over time. Much as the local variable x in e starts out pointing to 0 and eventually points to 1, the code of p starts out in encrypted form, and if/when any closure it returns is first applied, it changes to decrypted form. In both cases, it is critical that we never revert to the earlier state. Similarly, the closure returned by the evaluation of p starts out with its code pointer set to $\mathrm{bg} + 5$, but if/when the closure is ever applied, it will be set to $\mathrm{bg} + 13$. In this case, it is not so essential for correctness that the code pointer never revert back to $\mathrm{bg} + 5$, but it *is* essential that the closure's code pointer only be set to $\mathrm{bg} + 13$ when the code is in the decrypted state.

Given these restrictions on how the local state of p and e may evolve, it is but a short distance to a *bona fide* proof. Before sketching that proof, let us first review the recent work on Kripke logical relations that will put our reasoning on a solid footing.

3.3 Kripke Logical Relations and State Transition Systems

Logical relations are a well-established technique for reasoning about equivalence of higher-order programs. A logical relation is defined inductively on the type structure of the language: at base type the logical relation coincides with observable equality—*e.g.,* two programs of type int are logically related if they produce the same integer—and at higher type the relation is defined by interpreting each type operator by the appropriate logical connective—*e.g.,* two functions are related at type $\tau_1 \to \tau_2$ if relatedness of their arguments at type τ_1 *implies* relatedness of their results at type τ_2. The important feature about logical relations for our purposes is that they give considerable leeway to how related functions are implemented, so long as they produce related results. (As Benton and Hur [5] channeling Machiavelli put it, "the ends justify the means.")

In the presence of state, we cannot talk about the relatedness of two programs without making some assumptions and imposing some restrictions on how they manipulate state. This is where *Kripke logical relations* come in. Kripke logical relations are indexed by *possible worlds*, which represent a set of restrictions on the memories of the two programs under which the programs are guaranteed to behave equivalently. When we want to prove the relatedness of two programs under a world W, we suppose we are given arbitrary initial memories that are related by (*i.e.,* satisfy the restrictions of) W, and we proceed typically by showing that when evaluated under those memories the programs either (1) both diverge (don't terminate), or else (2) produce values and final memories that are related under some "future" world W' of W.

What does it mean for W' to be a future world of W? If in the course of evaluation the programs allocate fresh pieces of memory, W' may extend the initial world W with new restrictions governing the use of the freshly allocated memory. This approach allows us to establish whatever constraints we want on freshly allocated state that is kept *local*. (If the state is made globally accessible—*e.g.,* by being passed to the context at a ref type—then the state will have to obey the usual invariants dictated by the ref type.)

In traditional Kripke logical relations, such as those of Pitts and Stark [22], possible worlds essentially take the form of simple memory relations, *i.e.,* memory *invariants*. As we have seen in the "awkward" example, however, memory invariants are not necessarily enough; we need additionally the ability to describe assumptions about state that may change in a controlled and monotone way. (It is thus not a surprise that Pitts and Stark put forth the "awkward" example as an example for which their method was inadequate.)

To address this limitation, Ahmed *et al.* [2] proposed generalizing possible worlds to include the ability for a memory relation to *evolve*. Dreyer *et al.* [14] later streamlined and extended Ahmed *et al.*'s approach in various ways, and cast Ahmed *et al.*'s possible worlds as collections of *state transition systems (STS's)*. In the case of the "awkward" example, one can understand the restrictions placed on e's local variable x according to the following STS:

$$\begin{tikzcd} x \hookrightarrow 0 \arrow[r] & x \hookrightarrow 1 \end{tikzcd}$$

When this STS is first added to the initial world W, it starts out in the $x \hookrightarrow 0$ state, because after x is first allocated, it points to 0. However, under Dreyer *et al.*'s model, future worlds of W may not only place additional restrictions on fresh pieces of memory but also update the state of existing STS's in W. Thus, in some future world W', the above STS may be switched to the $x \hookrightarrow 1$ state, and memories satisfying W' would have to map x to 1. Furthermore, any future world of W' would have to remain in the $x \hookrightarrow 1$ state as there is no transition out of it. This corresponds to the intuitive reasoning about the example that we described in Section 3.1.

3.4 State Transition Systems for the Motivating Example

Using Kripke logical relations based on state transition systems, we can now roughly sketch the proof of equivalence of p and e.

We will prove that p and e are logically related in some initial world W_0 that includes some basic assumptions (in the form of STS's) about registers, the stack, etc. (See Section 7 for details.)

First, since we can assume p has just been loaded into memory, we can think of its code as a freshly allocated piece of memory, and we are therefore given the opportunity to extend W_0 with an STS governing p's code. For most programs, we would extend W_0 at this point with a one-state STS, representing the simple invariant that the code of the program never changes. For our motivating example, we instead extend W_0 with an STS of the form:

$$\begin{tikzcd} \text{encrypted} \arrow[r] & \text{decrypted} \end{tikzcd}$$

When the STS is in the left state, the associated memory relation will require that p's code be in its initial, encrypted form, and when the STS is in the right state, the memory relation will require that p's code be in decrypted form. Once decrypted, always decrypted.

When p and e are executed, the former allocates a fresh closure on the heap, setting its constituent code pointer to $\mathrm{bg} + 5$, and the latter allocates the local ref cell x, setting its contents to 0. Since both the closure and x are freshly allocated, we may at that point also extend the world with a new STS governing both of them:

$$\begin{tikzcd} \begin{array}{c} x \hookrightarrow 0 \ \wedge \\ \text{closure} \hookrightarrow \mathrm{bg} + 5 \end{array} \arrow[r] & \begin{array}{c} x \hookrightarrow 1 \ \wedge \\ \text{closure} \hookrightarrow \mathrm{bg} + 13 \end{array} \end{tikzcd}$$

We have joined the assumptions about them into one STS because they change in lockstep: when the functions returned by p and e are applied for the first time, x gets updated to 1 and the closure's code pointer gets updated to $\mathrm{bg} + 13$ simultaneously.

As noted in Section 3.2, it should never be the case that the code is still encrypted while the closure returned by p points to $\mathrm{bg} + 13$. (The closure would behave in an unspecified manner if it were called in such a state.) When adding the second STS, it is therefore important that we outlaw this possibility up front so that we will not have to consider it later. Fortunately, Dreyer *et al.*'s model, on which our logical relation is based, allows us to define the second STS in such a way that we can only be in its right state if the first STS is also in its right state.

Let W be the world resulting from extending W_0 with the above two STS's. What remains to be shown is that the functions returned by p and e are related in world W. So, suppose that W' is a future world of W, and that we begin executing the high- and low-level functions in some corresponding high- and low-level memories that are related by W'. There are three cases to consider, depending on the states of the two STS's of interest in W' (the fourth case was outlawed, as described above):

Case 1: Code is encrypted, $x \hookrightarrow 0$, closure $\hookrightarrow \mathrm{bg} + 5$. In this case, we first decrypt the code, and then set x to 1 and the closure's code pointer to $\mathrm{bg} + 13$.

Case 2: Code is decrypted, $x \hookrightarrow 0$, closure $\hookrightarrow \mathrm{bg} + 5$. In this case, we set x to 1 and the closure's code pointer to $\mathrm{bg} + 13$.

Case 3: Code is decrypted, $x \hookrightarrow 1$, closure $\hookrightarrow \mathrm{bg} + 13$. In this case, we set x to 1 and don't touch the closure's code pointer.

In all three cases, we end up transitioning to a future world W'' in which both STS's of interest are in the right state, if they weren't already there in W'. After making the state transition, both the high- and low-level functions invoke their callback arguments. Assuming they return, they will do so with memories that satisfy some future world of W'', but in the STS's of interest, there is nowhere to transition to. So we know that x must still point to 1. The high- and low-level functions must therefore both return the same result, namely 1, along with memories that satisfy a future world of the starting world W'.

3.5 Well-Bracketed State Changes and Private Transitions

For simplicity, we have glossed over many details in the above proof sketch. One important detail is how we reason about the stack. When the functions invoke their unknown callback arguments, we need to know that the callback will return the stack—*i.e.,* the contents of the stack segment up to the stack pointer sp—as it found it. One approach would be to bake this condition into our logical relation for functions. However, as we explained in the introduction, we have set out to define our relation in a largely language-generic fashion, and it would not make sense to bake a low-level property about the stack into a language-generic relation.

Instead, we wish to build this condition into the initial world under which we relate high- and low-level programs, but the property we desire of the stack is not expressible in terms of STS's as we have described them so far. To account for stack-like behavior, we employ another aspect of Dreyer *et al.*'s possible worlds, namely the idea of *private* vs. *public* transitions [14]. Private transitions were introduced in order to reason about so-called "well-bracketed" state changes, of which the behavior of the stack is a perfect example.

The basic idea is to label local state transitions as either public or private. Functions may make either private or public transitions internally, but viewed extensionally (*i.e.,* end-to-end), they must appear to make a public transition. In the STS that we use to reason about the stack, the states of the STS correspond to the possible states of the stack; every state is accessible from every other state by a private transition, but the only public transitions are self-transitions. This grants logically-related functions plenty of flexibility in how they manipulate the stack, but requires that, when they return, they leave the stack exactly how they found it.

We also use private transitions to reason about callee-save registers, which logically-related functions are expected to return as they found them, even if they modify them internally.

3.6 Reasoning in the Presence of Garbage Collection

Another important detail we have glossed over is how the presence of a garbage collector affects our proof of equivalence of p and e. Let us assume that we are using a standard mark-and-sweep or copying collector. The main effect such a collector has on our reasoning is that, whenever we pass control to the allocator (or to an unknown function that may call the allocator), we need to make sure that (1) all data that we care about being able to access in the future is in the reachable portion of the heap, and (2) there are no dangling pointers in the reachable portion of the heap.

Typically, guaranteeing these two conditions is straightforward. In our example, there is one call to the allocator (at $\mathrm{bg}+2$) and one call to an unknown function (at $\mathrm{bg}+17$). In the case of the former, there is nothing interesting to show. But in the case of the latter, it is important that when we increment the stack pointer before invoking the callback, we set the contents at the top of the stack to a value from which no dangling pointer may be reached. Here, we store in that stack slot the return address $\lfloor \mathrm{wk}_0 \rfloor$, which trivially satisfies this requirement.

So at an abstract level, reasoning in the presence of garbage collection is no big deal. What is more interesting is the technical question of how we actually implement this reasoning in the context of our logical relation. The central difficulty is that Kripke logical relations traditionally enjoy a *monotonicity* property, meaning that when two values are related in a world W, they are related in any future world of W. Monotonicity is essential when reasoning about unknown functions, such as the callback argument in our example. There, we were given the assumption that the callback argument was logically related in the world W', but we did not actually invoke it until we had transitioned to the future world W''. This step requires a use of monotonicity to show that the callback argument is still related in W''.

Unfortunately, garbage collection seems superficially to throw a wrench into monotonicity. For instance, in our logical relation we want to be able to relate a **HIGH** pair value $\langle v_1, v_2 \rangle$ with a memory location pointing to a representation of that pair value on the heap. How can we expect those two value representations to be related in all future worlds if, at some point in the future, the memory location may get deallocated (or its contents moved by a moving collector) and later reused for storing something else?

Our solution is to employ *logical memories*, which form a layer of abstraction over physical memories. Pointers in a logical memory are never moved or deallocated. Their connection to reality is established by a component of the logical memory called the *lookup table*, which specifies for any given logical pointer whether it is live and, if so, what physical pointer it corresponds to. We say that a logical memory $\mathbf{M}$ *represents* a physical memory Φ only if $\mathbf{M}$'s lookup table describes a bijection between the reachable portions of $\mathbf{M}$ and Φ.

We maintain a global invariant on the logical memory, which must hold before and after calls to the allocator, requiring that all reachable data be live (and thus not dangling) according to the lookup table. Together with the definition of what it means for a logical memory to represent a physical one, this invariant guarantees the allocator's precondition—(2) above—that there are no reachable dangling pointers. After the allocator returns, the lookup table of the logical memory may have completely changed—*e.g.,* due to a semi-space collection—but the only other change to the logical memory will be its extension with a freshly allocated logical pointer. Thus, if any data we care about was reachable prior to allocation—condition (1) above—it will still be reachable post-allocation, and by the global invariant it will also still be live.

With logical memories in hand, we can adapt our logical relations accordingly so that they relate values in **HIGH** with *logical values* (*i.e.,* logical pointers or non-pointer data) in **LOW**. We also define our possible worlds to impose invariants on *logical* memories, not physical ones. In this way, we regain monotonicity, as well as a clean, abstract account of memory locations that gives the garbage collector significant flexibility in how it implements them and allows us to essentially ignore how it does so.

4. A Language-Generic Kripke Logical Relation

Figure 3 defines a Kripke logical relation between two languages that is parameterized by abstract specifications for those languages ($\in$ LangSpec), as well as by a specification for the possible worlds relating the memories of those languages ($\in$ WorldSpec).

4.1 Language Specifications

A language specification (upper left of Figure 3) must provide sets of values (Val), computations (Com), continuations (Cont), memories (Mem), and configurations (Conf), together with a number of operations on these sets. (Note that we are assuming here a standard stratification on sets, and that Val, Com, etc. are "smaller" than LangSpec.) Most of these operations take elements of some of these sets and return a predicate on another of the sets. In most cases, this is because there may be a number of different representations of the same thing—*e.g.,* in **LOW**, a pair value $\langle v_1, v_2 \rangle$ is represented by a pointer that satisfies some conditions, but many pointers may satisfy those conditions.

"plugv" forms a configuration by plugging a value into a continuation under a given memory. "plugc" does the same, but plugging a computation instead of a value. "step" executes a configuration for one step of computation, resulting in either a new configuration, termination (*halt*), or failure (*fail*). "mdom" returns the domain of a memory, and "mdisj" takes two memories and returns a memory that contains their disjoint union, if it exists.

"oftype(τ)" determines whether a value is considered syntactically to have type τ under a certain memory. In our specification for **HIGH**, we include heap typings Σ in memories in order to define this predicate at ref type. For **LOW**, there is no notion of syntactic typing, but we find oftype convenient for expressing assumptions about the "syntactic" structure of closures (see Section 5).

$$
\begin{aligned}
&\mathrm{CType} \stackrel{\text{def}}{=} \{\, \tau \mid \mathrm{ftv}(\tau) = \emptyset \,\}\\
&\mathrm{LangSpec} \stackrel{\text{def}}{=}\\
&\{\, (\mathrm{Val}, \mathrm{Com}, \mathrm{Cont}, \mathrm{Mem}, \mathrm{Conf},\\
&\quad \mathrm{plugv}, \mathrm{plugc}, \mathrm{step}, \mathrm{mdom}, \mathrm{mdisj},\\
&\quad \mathrm{oftype}, \mathrm{base}_b, \mathrm{pair}, \mathrm{app}, \mathrm{appty},\\
&\quad \mathrm{pack}, \mathrm{roll}, \mathrm{ref}, \mathrm{asgn}) \mid\\
&\quad \mathrm{Val}, \mathrm{Com}, \mathrm{Cont}, \mathrm{Mem}, \mathrm{Conf} \in \mathrm{Set} \land\\
&\quad \mathrm{plugv} \in \mathrm{Val} \times \mathrm{Cont} \times \mathrm{Mem} \to \mathbb{P}(\mathrm{Conf}) \land\\
&\quad \mathrm{plugc} \in \mathrm{Com} \times \mathrm{Cont} \times \mathrm{Mem} \to \mathbb{P}(\mathrm{Conf}) \land\\
&\quad \mathrm{step} \in \mathrm{Conf} \to \mathrm{Conf} \uplus \{\, \mathit{fail}, \mathit{halt} \,\} \land\\
&\quad \mathrm{mdom} \in \mathrm{Mem} \to \mathbb{P}(\mathrm{Val}) \land\\
&\quad \mathrm{mdisj} \in \mathrm{Mem} \times \mathrm{Mem} \to \mathbb{P}(\mathrm{Mem}) \land\\
&\quad \mathrm{oftype} \in \mathrm{CType} \to \mathbb{P}(\mathrm{Val} \times \mathrm{Mem}) \land\\
&\quad \mathrm{base}_b \in \llbracket b \rrbracket \to \mathbb{P}(\mathrm{Val} \times \mathrm{Mem}) \land\\
&\quad \mathrm{pair} \in \mathrm{Val} \times \mathrm{Val} \to \mathbb{P}(\mathrm{Val} \times \mathrm{Mem}) \land\\
&\quad \mathrm{app} \in \mathrm{Val} \times \mathrm{Val} \to \mathbb{P}(\mathrm{Com}) \land\\
&\quad \mathrm{appty} \in \mathrm{Val} \times \mathrm{CType} \to \mathbb{P}(\mathrm{Com}) \land\\
&\quad \mathrm{pack} \in \mathrm{CType} \times \mathrm{Val} \to \mathbb{P}(\mathrm{Val} \times \mathrm{Mem}) \land\\
&\quad \mathrm{roll} \in \mathrm{Val} \to \mathbb{P}(\mathrm{Val} \times \mathrm{Mem}) \land\\
&\quad \mathrm{ref} \in \mathrm{Val} \to \mathbb{P}(\mathrm{Val} \times \mathrm{Mem}) \land\\
&\quad \mathrm{asgn} \in \mathrm{Mem} \times \mathrm{Val} \times \mathrm{Val} \rightharpoonup \mathrm{Mem} \land\\
&\quad \forall M_1, M_2.\, \forall M \in \mathrm{mdisj}(M_1, M_2).\\
&\qquad \mathrm{mdom}(M) \supseteq \mathrm{mdom}(M_1) \uplus \mathrm{mdom}(M_2) \,\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{For } \mathcal{L}_1, \mathcal{L}_2 \in \mathrm{LangSpec},\\
&\mathrm{WorldSpec} \stackrel{\text{def}}{=}\\
&\{\, (\mathrm{World}, \mathrm{lev}, \mathcal{M}, \mathcal{B}, \mathcal{O}, \triangleright, \sqsupseteq, \sqsupseteq_{\mathrm{pub}}) \mid\\
&\quad \mathrm{World} \in \mathrm{Set} \land\\
&\quad \mathrm{lev} \in \mathrm{World} \to \mathbb{N} \land\\
&\quad \mathcal{M} \in \mathrm{World} \to \mathbb{P}(\mathcal{L}_1.\mathrm{Mem} \times \mathcal{L}_2.\mathrm{Mem}) \land\\
&\quad \mathcal{B} \in \mathrm{World} \to \mathbb{P}(\mathcal{L}_1.\mathrm{Val} \times \mathcal{L}_2.\mathrm{Val}) \land\\
&\quad \mathcal{O} \in \mathrm{World} \to \mathbb{P}(\mathcal{L}_1.\mathrm{Conf} \times \mathcal{L}_2.\mathrm{Conf}) \land\\
&\quad \triangleright \in \mathrm{World} \to \mathrm{World} \land\\
&\quad \sqsupseteq \in \mathbb{P}(\mathrm{World} \times \mathrm{World}) \land\\
&\quad \sqsupseteq_{\mathrm{pub}} \in \mathbb{P}(\mathrm{World} \times \mathrm{World}) \land\\
&\quad \sqsupseteq, \sqsupseteq_{\mathrm{pub}} \text{ are preorders} \land \sqsupseteq_{\mathrm{pub}} \subseteq \sqsupseteq \ \land\\
&\quad \forall W' \sqsupseteq W.\, \triangleright W' \sqsupseteq \triangleright W \land\\
&\quad \forall W' \sqsupseteq_{\mathrm{pub}} W.\, \triangleright W' \sqsupseteq_{\mathrm{pub}} \triangleright W \land\\
&\quad \forall W.\, \triangleright W \sqsupseteq_{\mathrm{pub}} W \land\\
&\quad \forall W' \sqsupseteq W.\, \mathrm{lev}(W') \le \mathrm{lev}(W) \land\\
&\quad \forall W.\, \mathrm{lev}(W) > 0 \implies \mathrm{lev}(\triangleright W) = \mathrm{lev}(W) - 1 \,\}
\end{aligned}
$$

$$
\begin{aligned}
\sqsupset_{\triangleright} &\stackrel{\text{def}}{=} \{\, (W', W) \mid \mathrm{lev}(W) > 0 \land W' \sqsupseteq \triangleright W \,\}\\
\mathrm{WVRel} &\stackrel{\text{def}}{=} \{\, R \in \mathbb{P}(\mathrm{World} \times \mathcal{L}_1.\mathrm{Val} \times \mathcal{L}_2.\mathrm{Val}) \,\}\\
R(W) &\stackrel{\text{def}}{=} \{\, (\mathbf{v}_1, v_2) \mid (W, \mathbf{v}_1, v_2) \in R \,\}\\
\triangleright R &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \mid \mathrm{lev}(W) > 0 \implies (\triangleright W, \mathbf{v}_1, v_2) \in R \,\}\\
\square R &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \mid \forall W' \sqsupseteq W.\, (W', \mathbf{v}_1, v_2) \in R \,\}\\
R_{\sqsupset_{\triangleright} W} &\stackrel{\text{def}}{=} \{\, (W', \mathbf{v}_1, v_2) \mid W' \sqsupset_{\triangleright} W \land (W', \mathbf{v}_1, v_2) \in R \,\}\\
\overline{(R_1, R_2)} &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \mid \forall (\mathbf{M}_1, M_2) \in \mathcal{M}(W).\, (\mathbf{v}_1, \mathbf{M}_1) \in R_1 \land (v_2, M_2) \in R_2 \,\}\\
&\quad\ \text{for } R_1 \in \mathbb{P}(\mathcal{L}_1.\mathrm{Val} \times \mathcal{L}_1.\mathrm{Mem}), R_2 \in \mathbb{P}(\mathcal{L}_2.\mathrm{Val} \times \mathcal{L}_2.\mathrm{Mem})\\
\mathrm{TyValRel} &\stackrel{\text{def}}{=} \{\, (\tau_1, \tau_2, R) \mid \tau_1, \tau_2 \in \mathrm{CType} \land R \in \mathrm{WVRel} \,\}\\
\rho &\in \mathrm{TypeVar} \rightharpoonup \mathrm{TyValRel}\\
\rho.1(\tau) &\stackrel{\text{def}}{=} \tau[\rho(\alpha).\tau_1/\alpha] \qquad\qquad \rho.2(\tau) \stackrel{\text{def}}{=} \tau[\rho(\alpha).\tau_2/\alpha]\\
\mathrm{oftype}(\tau, \rho) &\stackrel{\text{def}}{=} \square\overline{(\mathcal{L}_1.\mathrm{oftype}(\rho.1(\tau)), \mathcal{L}_2.\mathrm{oftype}(\rho.2(\tau)))}\\
\mathcal{V}\llbracket \alpha \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\alpha, \rho) \mid (W, \mathbf{v}_1, v_2) \in \square\rho(\alpha).R \,\}\\
\mathcal{V}\llbracket b \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(b, \rho) \mid \exists x \in \llbracket b \rrbracket\,.\\
&\qquad (W, \mathbf{v}_1, v_2) \in \square\overline{(\mathcal{L}_1.\mathrm{base}_b(x), \mathcal{L}_2.\mathrm{base}_b(x))} \,\}\\
\mathcal{V}\llbracket \tau \times \tau' \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\tau \times \tau', \rho) \mid\\
&\qquad \exists (\mathbf{u}_1, u_2) \in \triangleright\mathcal{V}\llbracket \tau \rrbracket \rho(W).\, \exists (\mathbf{u}_1', u_2') \in \triangleright\mathcal{V}\llbracket \tau' \rrbracket \rho(W).\\
&\qquad (W, \mathbf{v}_1, v_2) \in \square\overline{(\mathcal{L}_1.\mathrm{pair}(\mathbf{u}_1, \mathbf{u}_1'), \mathcal{L}_2.\mathrm{pair}(u_2, u_2'))} \,\}\\
\mathcal{V}\llbracket \tau' \to \tau \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\tau' \to \tau, \rho) \mid \forall W' \sqsupset_{\triangleright} W.\, \forall (\mathbf{u}_1, u_2) \in \mathcal{V}\llbracket \tau' \rrbracket \rho(W').\\
&\qquad \forall \mathbf{e}_1 \in \mathcal{L}_1.\mathrm{app}(\mathbf{v}_1, \mathbf{u}_1).\, \forall e_2 \in \mathcal{L}_2.\mathrm{app}(v_2, u_2).\, (W', \mathbf{e}_1, e_2) \in \mathcal{E}\llbracket \tau \rrbracket \rho \,\}\\
\mathcal{V}\llbracket \forall\alpha.\, \tau \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\forall\alpha.\, \tau, \rho) \mid \forall W' \sqsupset_{\triangleright} W.\, \forall (\tau_1, \tau_2, R) \in \mathrm{TyValRel}.\\
&\qquad \forall \mathbf{e}_1 \in \mathcal{L}_1.\mathrm{appty}(\mathbf{v}_1, \tau_1).\, \forall e_2 \in \mathcal{L}_2.\mathrm{appty}(v_2, \tau_2).\\
&\qquad (W', \mathbf{e}_1, e_2) \in \mathcal{E}\llbracket \tau \rrbracket \rho[\alpha \mapsto (\tau_1, \tau_2, R)] \,\}\\
\mathcal{V}\llbracket \exists\alpha.\, \tau \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\exists\alpha.\, \tau, \rho) \mid\\
&\qquad \exists (\tau_1, \tau_2, R) \in \mathrm{TyValRel}.\, \exists (\mathbf{u}_1, u_2) \in \mathcal{V}\llbracket \tau \rrbracket \rho[\alpha \mapsto (\tau_1, \tau_2, R)](W).\\
&\qquad (W, \mathbf{v}_1, v_2) \in \square\overline{(\mathcal{L}_1.\mathrm{pack}(\tau_1, \mathbf{u}_1), \mathcal{L}_2.\mathrm{pack}(\tau_2, u_2))} \,\}\\
\mathcal{V}\llbracket \mathsf{ref}\, \tau \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\mathsf{ref}\, \tau, \rho) \mid \forall W' \sqsupseteq W.\, \forall (\mathbf{M}_1, M_2) \in \mathcal{M}(W').\\
&\quad (\mathbf{v}_1, v_2) \in \mathcal{B}(W') \land\\
&\quad \big(\exists (\mathbf{u}_1, u_2) \in \triangleright\mathcal{V}\llbracket \tau \rrbracket \rho(W').\, (\mathbf{v}_1, \mathbf{M}_1) \in \mathcal{L}_1.\mathrm{ref}(\mathbf{u}_1) \land (v_2, M_2) \in \mathcal{L}_2.\mathrm{ref}(u_2)\big) \land\\
&\quad \big(\forall (\mathbf{u}_1, u_2) \in \triangleright\mathcal{V}\llbracket \tau \rrbracket \rho(W').\, (\mathcal{L}_1.\mathrm{asgn}(\mathbf{M}_1, \mathbf{v}_1, \mathbf{u}_1), \mathcal{L}_2.\mathrm{asgn}(M_2, v_2, u_2)) \in \mathcal{M}(W')\big) \}\\
\mathcal{V}\llbracket \mu\alpha.\, \tau \rrbracket \rho &\stackrel{\text{def}}{=} \mu(F_{\alpha,\tau,\rho})\\
F_{\alpha,\tau,\rho} &\stackrel{\text{def}}{=} \lambda R.\, \{\, (W, \mathbf{v}_1, v_2) \in \mathrm{oftype}(\mu\alpha.\, \tau, \rho) \mid\\
&\qquad\quad \exists (\mathbf{u}_1, u_2) \in \mathcal{V}\llbracket \tau \rrbracket \rho[\alpha \mapsto (\rho.1(\mu\alpha.\, \tau), \rho.2(\mu\alpha.\, \tau), R)](W).\\
&\qquad\quad (W, \mathbf{v}_1, v_2) \in \square\overline{(\mathcal{L}_1.\mathrm{roll}(\mathbf{u}_1), \mathcal{L}_2.\mathrm{roll}(u_2))} \,\}\\
\mu(F)(W) &\stackrel{\text{def}}{=} F(\mu(F)_{\sqsupset_{\triangleright} W})(W)\\
\mathcal{K}\llbracket \tau \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{K}_1, K_2) \in \mathrm{World} \times \mathcal{L}_1.\mathrm{Cont} \times \mathcal{L}_2.\mathrm{Cont} \mid \forall W' \sqsupseteq_{\mathrm{pub}} W.\\
&\qquad \forall (\mathbf{v}_1, v_2) \in \mathcal{V}\llbracket \tau \rrbracket \rho(W').\, \forall (\mathbf{M}_1, M_2) \in \mathcal{M}(W').\\
&\qquad \forall C_1 \in \mathcal{L}_1.\mathrm{plugv}(\mathbf{v}_1, \mathbf{K}_1, \mathbf{M}_1).\, \forall C_2 \in \mathcal{L}_2.\mathrm{plugv}(v_2, K_2, M_2).\\
&\qquad (C_1, C_2) \in \mathcal{O}(W') \,\}\\
\mathcal{E}\llbracket \tau \rrbracket \rho &\stackrel{\text{def}}{=} \{\, (W, \mathbf{e}_1, e_2) \in \mathrm{World} \times \mathcal{L}_1.\mathrm{Com} \times \mathcal{L}_2.\mathrm{Com} \mid\\
&\qquad \forall (\mathbf{K}_1, K_2) \in \mathcal{K}\llbracket \tau \rrbracket \rho(W).\, \forall (\mathbf{M}_1, M_2) \in \mathcal{M}(W).\\
&\qquad \forall C_1 \in \mathcal{L}_1.\mathrm{plugc}(\mathbf{e}_1, \mathbf{K}_1, \mathbf{M}_1).\, \forall C_2 \in \mathcal{L}_2.\mathrm{plugc}(e_2, K_2, M_2).\\
&\qquad (C_1, C_2) \in \mathcal{O}(W) \,\}
\end{aligned}
$$

Figure 3. Language Specifications, World Specifications, and a Language-Generic Kripke Logical Relation

The remaining operations define the various syntactic language forms that are referenced in the definition of the logical relation. One point of note: determining whether a value represents a particular canonical form may require one to consider an accompanying memory. It is easy to see why—*e.g.,* one cannot determine if a pointer to a pair of heap cells represents $\langle v_1, v_2 \rangle$ without inspecting the heap. Determining whether a computation represents a function application also may require inspecting the memory, but we have found it technically more convenient to assume that any memory inspection is somehow built into the notion of computation (see Section 5 to see how this works in the **LOW** specification).

4.2 World Specifications

A world specification (lower left of Figure 3) defines a set of possible worlds relating the memories of two languages specified by $\mathcal{L}_1$ and $\mathcal{L}_2$, together with a variety of operations on and relations indexed by those worlds. In order to understand some of these, it is necessary to first say a word about our use of *step-indexing*.

Appel and McAllester [3] introduced step-indexing in order to model recursive types in foundational proof-carrying code. The basic idea is to use a natural number index ("step level") to stratify what would otherwise be a circular construction. Step-indexing has also proven useful in modeling other semantically circular notions like higher-order state, which is how we use them here. Due to space considerations, since our step-indexed construction follows closely that of [2] and [14], we refer the interested reader to those previous works for the relevant background.

In the world specification, the important step-indexing-related bits are the lev function, which returns the step level of a given world, and the $\triangleright$ (pronounced "later") operator, which returns an approximated version of the given world at one lower step level. We use $\triangleright$ to ensure well-foundedness of the logical relation.

"$\mathcal{M}(W)$" is the memory relation associated with W, which specifies when two memories (from $\mathcal{L}_1$ and $\mathcal{L}_2$) satisfy the constraints of the STS's in W. "$\mathcal{B}(W)$" is a bijection on values representing memory locations in $\mathcal{L}_1$ and $\mathcal{L}_2$. This is used in defining the logical relation for ref type. "$\mathcal{O}(W)$" is an observation relation on configurations. For the possible worlds we employ in this paper, $\mathcal{O}(W)$ actually only depends on $\mathrm{lev}(W)$, and it is defined to relate configurations that either both terminate (without failure) or that both run for at least $\mathrm{lev}(W)$ steps of computation. When we prove that two programs are logically related, we will prove it in starting worlds of an arbitrary step level, thus ensuring that the programs are observably equivalent for arbitrarily many computation steps.

Finally, "$\sqsupseteq$" defines the general "future world" relation between worlds, and "$\sqsupseteq_{\mathrm{pub}}$" defines a restricted "public" version of that relation: if $W' \sqsupseteq_{\mathrm{pub}} W$, then for any STS in W, the new state of that STS in W' must be accessible from its old state in W only by public transitions (see Section 3.5). Both $\sqsupseteq$ and $\sqsupseteq_{\mathrm{pub}}$ are preorders.

4.3 Kripke Logical Relation

The right side of Figure 3 displays our Kripke logical relation, whose definition is parametric w.r.t. $\mathcal{L}_1$, $\mathcal{L}_2$, and an instance of $\mathrm{WorldSpec}$ thereon. In the definition, we adopt the convention that the entities (values, continuations, etc.) from $\mathcal{L}_1$ appear in boldface ($\mathbf{v}$, $\mathbf{K}$, etc.) and the entities from $\mathcal{L}_2$ appear in italics (v, K, etc.). The coincidence of the notation for $\mathcal{L}_2$ entities with the notation for the corresponding entities from **HIGH** is deliberate, for in the next section we will instantiate $\mathcal{L}_1$ and $\mathcal{L}_2$ with our specifications for **LOW** and **HIGH**, respectively. We abuse notation in this way in order to avoid the proliferation of more than two fonts.

Our logical relation is based very closely on Dreyer *et al.*'s [14], with the principal difference being that the relevant linguistic forms have been abstracted away in the language specifications $\mathcal{L}_1$ and $\mathcal{L}_2$. For instance, in the logical relation for arrow types, we do not construct the applications $\mathbf{v}_1\mathbf{u}_1$ and $v_2 u_2$ directly, since $\mathcal{L}_1$ and $\mathcal{L}_2$ may not include an explicit application construct. Rather, we quantify over arbitrary computations $\mathbf{e}_1$ and e_2 drawn from $\mathcal{L}_1.\mathrm{app}(\mathbf{v}_1, \mathbf{u}_1)$ and $\mathcal{L}_2.\mathrm{app}(v_2, u_2)$, respectively.

The logical relation consists of a relation for values ($\mathcal{V}[\![\tau]\!]\rho$), one for continuations ($\mathcal{K}[\![\tau]\!]\rho$), and one for computations ($\mathcal{E}[\![\tau]\!]\rho$). Here, we assume that ρ is a relational interpretation of the free variables of τ, mapping them to arbitrary world-indexed value relations. For $\tau = \alpha$, $\mathcal{V}[\![\tau]\!]\rho$ is defined as the restriction of $\rho(\alpha)$ to triples $(W, \mathbf{v}_1, v_2)$ where $\mathbf{v}_1$ and v_2 are "well-typed" (according to $\mathcal{L}_i.\mathrm{oftype}$) and continue to be related in all future worlds of W. This last part, which is specified using the $\Box R$ operator defined at the top right of the figure, is key to ensuring monotonicity of the value relation. The pair and existential cases of the value relation also use the $\Box R$ operator in order to ensure monotonicity of data representations—*e.g.,* that if $\mathbf{v}_1$ represents a pair of $\mathbf{u}_1$ and $\mathbf{u}_1'$, it will continue to do so in all future worlds.

As in Appel *et al.* [4], the interpretation of recursive types is defined by induction on the "strictly future world" relation $\sqsupset_{\triangleright}$. This relation is well-founded because $W' \sqsupset_{\triangleright} W$ implies that W' has a lower step level than W. By defining the recursive type case this way, we can relate **HIGH** programs, where roll and unroll are explicit coercions, to **LOW** programs where they have been erased.

The ref type case relates two memory locations if dereferencing, assigning and testing them for pointer equality will always produce related results. The condition on pointer equality testing is guaranteed by the requirement that the locations be in the bijection of the world in which they are related.

The value relation is lifted to a relation on computations by the technique of *biorthogonality* (aka $\top\top$*-closure*) [22, 17]. The idea is to define two computations to be related if they behave in an observably equivalent manner when plugged into related continuations. Two continuations are in turn related if they behave in an observably equivalent manner when plugged with related values. By quantifying only over public future worlds in the definition of $\mathcal{K}[\![\tau]\!]\rho$, we ensure that computations may only make public transitions when viewed end-to-end, as per the discussion in Section 3.5.

This mind-bending technique is well-suited to languages where the evaluation of a computation is *context-sensitive* in the sense that it cannot be performed in ignorance of its continuation. Such is the case with **LOW**, where a computation always ends with a jump to its return address. As we shall see, the fact that the return address really is a valid return address, which is part of the contract between computation and continuation, will be encoded in the **LOW** implementation of plugc that we define in Section 5.

5. Implementing the Specifications

In order to instantiate our logical relations to relate **LOW** and **HIGH** entities, we must first show how to implement the abstract $\mathrm{LangSpec}$ interface for both languages.

For **HIGH**, the implementation of the interface is almost entirely straightforward, as all the required entities (values, computations) have direct correspondents in the **HIGH** language. The only slightly unusual bit is that we define **HIGH** memories to be pairs of heaps *and* heap typings Σ. The inclusion of the heap typing is necessary for defining oftype. For the remaining details, please see the companion technical appendix [15].

Low-Level Entities The implementation of $\mathrm{LangSpec}$ for **LOW** (Figure 4) is much more interesting. As described in Section 3.6, we employ a notion of *logical values* $\mathbf{v}$, which are either non-pointer words $\underline{w}$ or logical pointers $\widehat{\boldsymbol{l}}$. Logical Lvalues are similar to physical PLvalues except that logical Lvalues include logical heap locations $\langle \boldsymbol{l} : o\rangle_{\mathrm{h}}$ in place of physical ones $\langle a\rangle_{\mathrm{h}}$. We include the offset o because the logical heap is (for convenience) modeled two-dimensionally as a list of blocks.

LOW computations $\mathbf{e}$ are 4-tuples $(\mathrm{cpc}, \mathrm{kpc}, \mathrm{vloc}, \mathrm{data})$, where: cpc is the code address where the computation begins, kpc is the return address, vloc is the Lvalue where the return value will be stored, and data is a memory predicate that must be satisfied in order for the computation to be correctly executed. **LOW** continuations $\mathbf{K}$ are pairs $(\mathrm{kpc}, \mathrm{vloc})$, where: kpc is the code address where the continuation begins, and vloc is the Lvalue where the input to the continuation should be placed. It is worth noting that, unlike $\mathbf{e}.\mathrm{cpc}$ and $\mathbf{e}.\mathrm{kpc}$, the kpc in $\mathbf{K}$ must be a code address, not an Rvalue, because when we plug an $\mathbf{e}$ into a $\mathbf{K}$, we need to know that the value of $\mathbf{K}.\mathrm{kpc}$ will be the same before and after the execution of $\mathbf{e}$. Were $\mathbf{K}.\mathrm{kpc}$ an Rvalue, we would not know that.

Logical memories $\mathbf{M}$ are 6-tuples $(\mathrm{code}, \mathrm{reg}, \mathrm{stk}, \mathrm{hp}, \mathrm{tab}, \mathrm{shp})$, where: code is the code segment, reg is the register file (minus sp since it is determined by the size of the stack), stk is the stack, hp is the heap, tab is the lookup table (described in Section 3.6), and shp is the *system heap*, which is a separate portion of the heap controlled by the runtime system. The lookup table maps each logical pointer to a physical pointer and the size of the memory block starting at that address. The pointer is live iff the size is > 0. Note that reg, stk, and hp are all maps from various Lvalues to *logical* values, whereas the tab and shp are maps to physical values. In the proofs, we end up treating tab and shp as essentially black boxes, since they can be changed at whim by the allocator, whereas the allocator should not mess around with the logical portion of the memory represented by reg, stk, and hp.

Lastly, note that reg, stk, tab, and shp may all be undefined (undef). This is a useful technical device for defining the disjoint union of several partial memories: it enables us to specify that only one of those partial memories contains information about, say, the stack. (See the definition of "mdisj" below.)

Low-Level Representations of High-Level Constructs The bottom left of Figure 4 defines **LOW** representations of various high-level constructs, as required by the $\mathrm{LangSpec}$ interface. A pair of $\mathbf{v}_1$ and $\mathbf{v}_2$ is represented as a pointer to a pair of cells containing $\mathbf{v}_1$

$$\begin{array}{lll}
\text{Loc} & \stackrel{\text{def}}{=} & \{\, \boldsymbol{l} \in \mathbb{N} \,\} \\
\text{Word} & \stackrel{\text{def}}{=} & \{\, w \in \mathbb{N} \,\} \\
\mathbf{v} \in \text{Val} & ::= & \underline{w} \mid \widehat{\boldsymbol{l}} \\
\text{lv} \in \text{Lvalue} & ::= & \lfloor r \rfloor \mid \langle a \rangle_{\text{s}} \mid \langle r - o \rangle_{\text{s}} \mid \langle \boldsymbol{l} : o \rangle_{\text{h}} \mid \langle r + o \rangle_{\text{h}} \\
\text{rv} \in \text{Rvalue} & ::= & \text{lv} \mid \mathbf{v} \\
\text{Com} & \stackrel{\text{def}}{=} & \{\, \mathbf{e} = (\text{cpc}, \text{kpc}, \text{vloc}, \text{data}) \\
 & & \quad \in \text{Rvalue} \times \text{Rvalue} \times \text{Lvalue} \times \mathbb{P}(\text{Mem}) \,\} \\
\text{Cont} & \stackrel{\text{def}}{=} & \{\, \mathbf{K} = (\text{kpc}, \text{vloc}) \in \text{PAddr} \times \text{Lvalue} \,\} \\
\text{CodeFrag} & \stackrel{\text{def}}{=} & \text{PAddr} \rightharpoonup_{\text{fin}} \text{Instruction} \\
\text{RegFile} & \stackrel{\text{def}}{=} & (\text{Register} \setminus \{\text{sp}\} \to \text{Val}) \uplus \{\, \text{undef} \,\} \\
\text{List } X & \stackrel{\text{def}}{=} & \{\, (x_0, \ldots, x_{n-1}) \mid n \in \mathbb{N} \wedge x_0, \ldots, x_{n-1} \in X \,\} \\
\text{Stack} & \stackrel{\text{def}}{=} & \text{List Val} \uplus \{\, \text{undef} \,\} \\
\text{Heap} & \stackrel{\text{def}}{=} & \text{Loc} \rightharpoonup_{\text{fin}} \text{List Val} \\
\text{Table} & \stackrel{\text{def}}{=} & (\text{Loc} \rightharpoonup_{\text{fin}} \mathbb{N} \times \text{PAddr}) \uplus \{\, \text{undef} \,\} \\
\text{SysHeap} & \stackrel{\text{def}}{=} & (\text{PAddr} \rightharpoonup \text{Word}) \uplus \{\, \text{undef} \,\} \\
\text{Mem} & \stackrel{\text{def}}{=} & \{\, \mathbf{M} = (\text{code}, \text{reg}, \text{stk}, \text{hp}, \text{tab}, \text{shp}) \\
 & & \in \text{CodeFrag} \times \text{RegFile} \times \text{Stack} \times \text{Heap} \times \text{Table} \times \text{SysHeap} \,\} \\
\text{Conf} & \stackrel{\text{def}}{=} & \text{PConf} \\
\text{oftype}(\tau) & \stackrel{\text{def}}{=} & \{\, (\mathbf{v}, \mathbf{M}) \in \text{Val} \times \text{Mem} \mid \\
 & & \quad \forall \tau_1, \tau_2.\, \tau = \tau_1 \to \tau_2 \implies \exists \boldsymbol{l}, w.\, \mathbf{v} = \widehat{\boldsymbol{l}} \wedge \mathbf{M}.\text{hp}(\boldsymbol{l})(0) = \underline{w} \wedge \\
 & & \quad \forall \alpha, \tau'.\, \tau = \forall \alpha.\, \tau' \implies \exists \boldsymbol{l}, w.\, \mathbf{v} = \widehat{\boldsymbol{l}} \wedge \mathbf{M}.\text{hp}(\boldsymbol{l})(0) = \underline{w} \,\} \\
\text{base}_b(x) & \stackrel{\text{def}}{=} & \{\, (\mathbf{v}, \mathbf{M}) \in \text{Val} \times \text{Mem} \mid \mathbf{v} \text{ is a representation of } x \,\} \\
\text{pair}(\mathbf{v}_1, \mathbf{v}_2) & \stackrel{\text{def}}{=} & \{\, (\mathbf{v}, \mathbf{M}) \in \text{Val} \times \text{Mem} \mid \exists \boldsymbol{l}.\, \mathbf{v} = \widehat{\boldsymbol{l}} \wedge \\
 & & \quad \mathbf{M}.\text{hp}(\boldsymbol{l})(0) = \mathbf{v}_1 \wedge \mathbf{M}.\text{hp}(\boldsymbol{l})(1) = \mathbf{v}_2 \,\} \\
\text{app}(\mathbf{v}_1, \mathbf{v}_2) & \stackrel{\text{def}}{=} & \{\, \mathbf{e} \in \text{Com} \mid \exists \boldsymbol{l}.\, \mathbf{v}_1 = \widehat{\boldsymbol{l}} \wedge \\
 & & \quad \mathbf{e}.\text{cpc} = \langle \boldsymbol{l} : 0 \rangle_{\text{h}} \wedge \mathbf{e}.\text{kpc} = \lfloor \text{wk}_0 \rfloor \wedge \mathbf{e}.\text{vloc} = \lfloor \text{wk}_5 \rfloor \wedge \\
 & & \quad \mathbf{e}.\text{data} = \{\, \mathbf{M} \in \text{Mem} \mid \mathbf{M}.\text{reg}(\text{wk}_1) = \mathbf{v}_1 \wedge \mathbf{M}.\text{reg}(\text{wk}_2) = \mathbf{v}_2 \,\} \,\} \\
\text{appty}(\mathbf{v}, \tau) & \stackrel{\text{def}}{=} & \{\, \mathbf{e} \in \text{Com} \mid \exists \boldsymbol{l}.\, \mathbf{v} = \widehat{\boldsymbol{l}} \wedge \\
 & & \quad \mathbf{e}.\text{cpc} = \langle \boldsymbol{l} : 0 \rangle_{\text{h}} \wedge \mathbf{e}.\text{kpc} = \lfloor \text{wk}_0 \rfloor \wedge \mathbf{e}.\text{vloc} = \lfloor \text{wk}_5 \rfloor \wedge \\
 & & \quad \mathbf{e}.\text{data} = \{\, \mathbf{M} \in \text{Mem} \mid \mathbf{M}.\text{reg}(\text{wk}_1) = \mathbf{v} \,\} \,\} \\
\text{pack}(\tau, \mathbf{v}) & \stackrel{\text{def}}{=} & \{\, (\mathbf{v}', \mathbf{M}) \in \text{Val} \times \text{Mem} \mid \mathbf{v}' = \mathbf{v} \,\} \\
\text{roll}(\mathbf{v}) & \stackrel{\text{def}}{=} & \{\, (\mathbf{v}', \mathbf{M}) \in \text{Val} \times \text{Mem} \mid \mathbf{v}' = \mathbf{v} \,\} \\
\text{ref}(\mathbf{v}) & \stackrel{\text{def}}{=} & \{\, (\mathbf{v}', \mathbf{M}) \in \text{Val} \times \text{Mem} \mid \exists \boldsymbol{l}.\, \mathbf{v}' = \widehat{\boldsymbol{l}} \wedge \mathbf{M}.\text{hp}(\boldsymbol{l})(0) = \mathbf{v} \,\} \\
\text{asgn}(\mathbf{M}, \mathbf{v}_1, \mathbf{v}_2) & \stackrel{\text{def}}{=} & \begin{cases} \mathbf{M}[\boldsymbol{l} : 0 \mapsto \mathbf{v}_2]_{\text{hp}} & \text{if } \mathbf{v}_1 = \widehat{\boldsymbol{l}} \wedge |\mathbf{M}.\text{hp}(\boldsymbol{l})| > 0 \\ \text{undef} & \text{otherwise} \end{cases}
\end{array}$$

$$|\mathbf{v}| \stackrel{\text{def}}{=} \begin{cases} w & \text{if } \mathbf{v} = \underline{w} \\ \boldsymbol{l} & \text{if } \mathbf{v} = \widehat{\boldsymbol{l}} \end{cases}$$

$$\begin{array}{llll}
\mathbf{M}(\mathbf{v}) & \stackrel{\text{def}}{=} \mathbf{v} & \mathbf{M}(\lfloor r \rfloor) & \stackrel{\text{def}}{=} \mathbf{M}.\text{reg}(r) \\
\mathbf{M}(\langle a \rangle_{\text{s}}) & \stackrel{\text{def}}{=} \mathbf{M}.\text{stk}(a) & \mathbf{M}(\langle r - o \rangle_{\text{s}}) & \stackrel{\text{def}}{=} \mathbf{M}.\text{stk}(|\mathbf{M}.\text{reg}(r)| - o) \\
\mathbf{M}(\langle \boldsymbol{l} : o \rangle_{\text{h}}) & \stackrel{\text{def}}{=} \mathbf{M}.\text{hp}(\boldsymbol{l})(o) & \mathbf{M}(\langle r + o \rangle_{\text{h}}) & \stackrel{\text{def}}{=} \mathbf{M}.\text{hp}(|\mathbf{M}.\text{reg}(r)|)(o)
\end{array}$$

$$\begin{array}{lll}
\mathbf{M}[[T, S]] & \stackrel{\text{def}}{=} & (\mathbf{M}.\text{code}, \mathbf{M}.\text{reg}, \mathbf{M}.\text{stk}, \mathbf{M}.\text{hp}, T, S) \\
\mathbf{M}[\boldsymbol{l} : o \mapsto \mathbf{v}]_{\text{hp}} & \stackrel{\text{def}}{=} & (\mathbf{M}.\text{code}, \mathbf{M}.\text{reg}, \mathbf{M}.\text{stk}, \\
 & & \quad \mathbf{M}.\text{hp}[\boldsymbol{l} \mapsto (\mathbf{v}_0, \ldots, \mathbf{v}_{o-1}, \mathbf{v}, \mathbf{v}_{o+1}, \ldots, \mathbf{v}_{n-1})], \\
 & & \quad \mathbf{M}.\text{tab}, \mathbf{M}.\text{shp}) \\
 & & \quad \text{if } \mathbf{M}.\text{hp}(\boldsymbol{l}) = (\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}) \wedge o < n \\
\text{phyv}(\mathbf{M})(\mathbf{v}) & \stackrel{\text{def}}{=} & \begin{cases} \underline{w} & \text{if } \mathbf{v} = \underline{w} \\ \widehat{a} & \text{if } \mathbf{v} = \widehat{\boldsymbol{l}} \wedge \mathbf{M}.\text{tab}(\boldsymbol{l}) = (n, a) \\ \text{undef} & \text{otherwise} \end{cases} \\
\text{phyh}(\mathbf{M}) & \stackrel{\text{def}}{=} & \biguplus_{\mathbf{M}.\text{tab}(\boldsymbol{l}) = (n,a) \,\wedge\, n > 0 \,\wedge\, \mathbf{M}.\text{hp}(\boldsymbol{l}) = (\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})} [a \mapsto \text{phyv}(\mathbf{M})(\mathbf{v}_0), \ldots, \text{phyv}(\mathbf{M})(\mathbf{v}_{n-1})] \\
\mathbf{M} \text{ repr } \Phi & \stackrel{\text{def}}{=} & \Phi.\text{code} \supseteq \mathbf{M}.\text{code} \wedge \\
 & & \Phi.\text{reg} \supseteq \text{phyv}(\mathbf{M}) \circ \mathbf{M}.\text{reg} \wedge \Phi.\text{reg}(\text{sp}) = \underline{|\mathbf{M}.\text{stk}|} \wedge \\
 & & \forall j < |\mathbf{M}.\text{stk}|.\, \Phi.\text{stk}(j) = \text{phyv}(\mathbf{M})(\mathbf{M}.\text{stk}(j)) \wedge \\
 & & \Phi.\text{hp} \supseteq \text{phyh}(\mathbf{M}) \uplus \mathbf{M}.\text{shp} \wedge \\
 & & \forall \boldsymbol{l}, n, a.\, \mathbf{M}.\text{tab}(\boldsymbol{l}) = (n, a) \wedge n > 0 \implies |\mathbf{M}.\text{hp}(\boldsymbol{l})| = n \\
\text{plugv}(\mathbf{v}, \mathbf{K}, \mathbf{M}) & \stackrel{\text{def}}{=} & \{\, (\Phi, \text{pc}) \in \text{Conf} \mid \mathbf{M} \text{ repr } \Phi \wedge \\
 & & \quad \text{pc} = \mathbf{K}.\text{kpc} \wedge \mathbf{M}(\mathbf{K}.\text{vloc}) = \mathbf{v} \,\} \\
\text{plugc}(\mathbf{e}, \mathbf{K}, \mathbf{M}) & \stackrel{\text{def}}{=} & \{\, (\Phi, \text{pc}) \in \text{Conf} \mid \mathbf{M} \text{ repr } \Phi \wedge \mathbf{M} \in \mathbf{e}.\text{data} \wedge \\
 & & \underline{\text{pc}} = \mathbf{M}(\mathbf{e}.\text{cpc}) \wedge \mathbf{M}(\mathbf{e}.\text{kpc}) = \underline{\mathbf{K}.\text{kpc}} \wedge \mathbf{e}.\text{vloc} = \mathbf{K}.\text{vloc} \,\} \\
\text{step}(\Phi, \text{pc}) & \stackrel{\text{def}}{=} & R \quad \text{with } (\Phi, \text{pc}) \hookrightarrow R \\
\text{mdom}(\mathbf{M}) & \stackrel{\text{def}}{=} & \{\, \widehat{\boldsymbol{l}} \in \text{Val} \mid \boldsymbol{l} \in \text{dom}(\mathbf{M}.\text{hp}) \,\} \\
\text{mdisj}(\mathbf{M}_1, \mathbf{M}_2) & \stackrel{\text{def}}{=} & \{\, \mathbf{M} \in \text{Mem} \mid \\
 & & \quad \mathbf{M}.\text{code} \supseteq \mathbf{M}_1.\text{code} \uplus \mathbf{M}_2.\text{code} \wedge \\
 & & \quad \mathbf{M}.\text{hp} \supseteq \mathbf{M}_1.\text{hp} \uplus \mathbf{M}_2.\text{hp} \wedge \\
 & & \quad \text{nosh}(\mathbf{M}.\text{reg}, \mathbf{M}_1.\text{reg}, \mathbf{M}_2.\text{reg}) \wedge \\
 & & \quad \text{nosh}(\mathbf{M}.\text{stk}, \mathbf{M}_1.\text{stk}, \mathbf{M}_2.\text{stk}) \wedge \\
 & & \quad \text{nosh}(\mathbf{M}.\text{tab}, \mathbf{M}_1.\text{tab}, \mathbf{M}_2.\text{tab}) \wedge \\
 & & \quad \text{nosh}(\mathbf{M}.\text{shp}, \mathbf{M}_1.\text{shp}, \mathbf{M}_2.\text{shp}) \,\} \\
\text{nosh}(X, X_1, X_2) & \stackrel{\text{def}}{=} & (X_1 \neq \text{undef} \implies X_2 = \text{undef} \wedge X = X_1) \wedge \\
 & & (X_2 \neq \text{undef} \implies X_1 = \text{undef} \wedge X = X_2)
\end{array}$$

Figure 4. The Implementation of LangSpec for **LOW**

and $\mathbf{v}_2$. $\text{pack}(\tau, \mathbf{v})$ and $\text{roll}(\mathbf{v})$ are represented the same as $\mathbf{v}$. References are represented directly as pointers. We also use $\text{oftype}(\tau)$ to enforce that values of arrow and universal type are represented by pointers to closures whose first cell is *not* a logical pointer. (This ensures that we can jump to the code address directly.)

The most interesting bit is the representation of $\text{app}(\mathbf{v}_1, \mathbf{v}_2)$ (and $\text{appty}(\mathbf{v}, \tau)$, which is similar). In order for a computation $\mathbf{e}$ to represent this application, $\mathbf{v}_1$ is assumed to be a pointer $\widehat{\boldsymbol{l}}$ to a closure. The starting address of the function application ($\mathbf{e}.\text{cpc}$) is thus taken to be the code address stored in the first cell of the closure, *i.e.,* $\langle \boldsymbol{l} : 0 \rangle_{\text{h}}$. Our calling convention is that, when the function is called, the return address should be stored in wk_0, and when the function returns, the return value should be stored in wk_5, so $\mathbf{e}.\text{kpc}$ and $\mathbf{e}.\text{vloc}$ reflect this convention. Finally, the memory predicate $\mathbf{e}.\text{data}$ requires that when control is passed to $\mathbf{e}.\text{cpc}$, the function $\mathbf{v}_1$ and argument $\mathbf{v}_2$ are stored in wk_1 and wk_2.

Connecting Logical and Physical Memories The right side of Figure 4 defines the remaining elements of LangSpec, along with a number of auxiliary operations. The operations at the top right give shorthand for various lookup and update operations on logical memories. "$\text{phyv}(\mathbf{M})(\mathbf{v})$" returns the physical interpretation of $\mathbf{v}$ according to $\mathbf{M}$'s lookup table, if one exists, and "$\text{phyh}(\mathbf{M})$" returns the live portion of the physical heap according to $\mathbf{M}$'s lookup table. Note that the definition of phyh demands that the physical representations of logical memory blocks with distinct head pointers be disjoint, thus ensuring a proper bijection between the reachable parts of the physical and logical heaps.

"$\mathbf{M}$ repr Φ" says that $\mathbf{M}$ is a valid logical abstraction of Φ. The definition is fairly straightforward, making use of phyv and phyh as one would expect. The fourth line of the definition guarantees that the reachable heap is disjoint from the system heap, and the fifth condition just checks that the block sizes specified for live data in $\mathbf{M}.\text{tab}$ are correct. Note that Φ may contain arbitrary other junk (in the code, stack, and heap segments) that is not described by $\mathbf{M}$.

Plugging Continuations Using $\mathbf{M}$ repr Φ, it is easy to specify how to plug continuations with values and computations. A configuration (Φ, pc) belongs to $\text{plugv}(\mathbf{v}, \mathbf{K}, \mathbf{M})$ if (1) $\mathbf{M}$ is a valid abstraction of Φ, (2) the program counter pc is set to the starting address of the continuation ($\mathbf{K}.\text{kpc}$), and (3) the value $\mathbf{v}$ is stored in the location where the continuation is expecting it ($\mathbf{K}.\text{vloc}$).

A configuration (Φ, pc) belongs to $\text{plugc}(\mathbf{e}, \mathbf{K}, \mathbf{M})$ if (1) $\mathbf{M}$ is a valid abstraction of Φ, (2) $\mathbf{M}$ satisfies the memory constraints demanded by $\mathbf{e}$, (3) the program counter pc is set to the starting address of the computation ($\mathbf{e}.\text{cpc}$), (4) the starting address of the continuation ($\mathbf{K}.\text{kpc}$) is stored in the place where the computation is expecting to find its return address ($\mathbf{e}.\text{kpc}$), and (5) the place where $\mathbf{e}$ will store its return value ($\mathbf{e}.\text{vloc}$) is the same place where $\mathbf{K}$ is expecting to find its input value ($\mathbf{K}.\text{vloc}$).

The remaining definitions (of step, mdom, and mdisj) are fairly self-explanatory. As mentioned earlier, the definition of

$$
\begin{aligned}
&[\mathrm{bg} \Mapsto \mathrm{instrs}] \stackrel{\mathrm{def}}{=} [\mathrm{bg} \mapsto \mathrm{instrs}(0), \ldots, \mathrm{instrs}(|\mathrm{instrs}| - 1)] \\
&\{C\}_{\mathrm{code}} \stackrel{\mathrm{def}}{=} (C, \mathrm{undef}, \mathrm{undef}, \emptyset, \mathrm{undef}, \mathrm{undef}) \in \mathrm{Mem} \\
&\{H\}_{\mathrm{heap}} \stackrel{\mathrm{def}}{=} (\emptyset, \mathrm{undef}, \mathrm{undef}, H, \mathrm{undef}, \mathrm{undef}) \in \mathrm{Mem} \\
&\mathbf{v} \text{ live in } \mathbf{M} \stackrel{\mathrm{def}}{=} \begin{cases} \top & \text{if } \mathbf{v} = \underline{w} \\ \exists n, a.\, \mathbf{M}.\mathrm{tab}(\boldsymbol{l}) = (n, a) \wedge n > 0 & \text{if } \mathbf{v} = \widehat{\boldsymbol{l}} \end{cases} \\
&\mathrm{reach}_0(\mathbf{M}) \stackrel{\mathrm{def}}{=} \{\, \boldsymbol{l} \mid \exists r \in \mathrm{Register}.\, \widehat{\boldsymbol{l}} = \mathbf{M}.\mathrm{reg}(r) \,\} \cup \\
&\qquad \{\, \boldsymbol{l} \mid \exists j < |\mathbf{M}.\mathrm{stk}|.\, \widehat{\boldsymbol{l}} = \mathbf{M}.\mathrm{stk}(j) \,\} \\
&\mathrm{reach}_{i+1}(\mathbf{M}) \stackrel{\mathrm{def}}{=} \mathrm{reach}_i(\mathbf{M}) \cup \\
&\qquad \{\, \boldsymbol{l} \mid \exists \boldsymbol{l}' \in \mathrm{reach}_i(\mathbf{M}).\, \exists j.\, \widehat{\boldsymbol{l}} = \mathbf{M}.\mathrm{hp}(\boldsymbol{l}')(j) \,\} \\
&\mathrm{reach}(\mathbf{M}) \stackrel{\mathrm{def}}{=} \textstyle\bigcup_{i \in \mathbb{N}} \mathrm{reach}_i(\mathbf{M}) \\
&\mathrm{AllocSpec} \stackrel{\mathrm{def}}{=} \\
&\{\, \mathcal{A} \in \mathrm{PAddr} \to \{\, (\mathrm{init}, \mathrm{alloc}, \mathrm{instrs}, I) \\
&\qquad\qquad \in \mathrm{PAddr} \times \mathrm{PAddr} \times \mathrm{List\ Instruction} \times \\
&\qquad\qquad\quad \mathbb{P}(\mathrm{Table} \times \mathrm{SysHeap}) \,\} \mid \\
&\ \forall \mathrm{gcbg}, \Phi, \mathrm{pc}. \\
&\quad \Phi.\mathrm{code} \supseteq [\mathrm{gcbg} \Mapsto \mathcal{A}(\mathrm{gcbg}).\mathrm{instrs}] \wedge \Phi.\mathrm{reg}(\mathrm{wk}_4) = \underline{\mathrm{pc}} \implies \\
&\quad \exists \mathbf{M}', \Phi'. \\
&\qquad (\Phi, \mathcal{A}(\mathrm{gcbg}).\mathrm{init}) \stackrel{*}{\hookrightarrow} (\Phi', \mathrm{pc}) \wedge \\
&\qquad \Phi'.\mathrm{code} = \Phi.\mathrm{code} \wedge \mathbf{M}'.\mathrm{code} = [\mathrm{gcbg} \Mapsto \mathcal{A}(\mathrm{gcbg}).\mathrm{instrs}] \wedge \\
&\qquad \mathbf{M}' \text{ repr } \Phi' \wedge \mathbf{M}' \in \mathcal{A}.GR(\mathrm{gcbg}) \wedge \mathbf{M}' \in \mathcal{A}.MR(\mathrm{gcbg}) \wedge \\
&\ \forall \mathrm{gcbg}, \mathbf{M}, \Phi, \mathrm{pc}, n. \\
&\quad \mathbf{M} \text{ repr } \Phi \wedge \mathbf{M} \in \mathcal{A}.GR(\mathrm{gcbg}) \wedge \mathbf{M} \in \mathcal{A}.MR(\mathrm{gcbg}) \wedge \\
&\quad \mathbf{M}.\mathrm{reg}(\mathrm{wk}_4) = \underline{\mathrm{pc}} \wedge \mathbf{M}.\mathrm{reg}(\mathrm{wk}_5) = \underline{n} \implies \\
&\quad \exists \Phi', \mathbf{M}', T, S, w, \overline{\boldsymbol{l}}, w_0, \ldots, w_{n-1}. \\
&\qquad (\Phi, \mathcal{A}(\mathrm{gcbg}).\mathrm{alloc}) \stackrel{*}{\hookrightarrow} (\Phi', \mathrm{pc}) \wedge \\
&\qquad \mathbf{M}' \text{ repr } \Phi' \wedge \mathbf{M}' \in \mathcal{A}.GR(\mathrm{gcbg}) \wedge \mathbf{M}' \in \mathcal{A}.MR(\mathrm{gcbg}) \wedge \\
&\qquad \mathbf{M}' = \mathbf{M}[[T, S]][\mathrm{wk}_4 \mapsto \underline{w}]_{\mathrm{reg}}[\mathrm{wk}_5 \mapsto \widehat{\boldsymbol{l}}]_{\mathrm{reg}} \uplus \\
&\qquad\qquad \{[\boldsymbol{l} \mapsto (\underline{w_0}, \ldots, \underline{w_{n-1}})]\}_{\mathrm{heap}} \,\} \\
&\mathcal{A}.GR(\mathrm{gcbg}) \stackrel{\mathrm{def}}{=} \{\, \mathbf{M} \in \mathrm{Mem} \mid \forall \boldsymbol{l} \in \mathrm{reach}(\mathbf{M}).\, \widehat{\boldsymbol{l}} \text{ live in } \mathbf{M} \,\} \\
&\mathcal{A}.MR(\mathrm{gcbg}) \stackrel{\mathrm{def}}{=} \{\, \mathbf{M} \in \mathrm{Mem} \mid (\mathbf{M}.\mathrm{tab}, \mathbf{M}.\mathrm{shp}) \in \mathcal{A}(\mathrm{gcbg}).I \wedge \\
&\qquad\qquad\qquad\qquad \mathbf{M}.\mathrm{code} \supseteq [\mathrm{gcbg} \Mapsto \mathcal{A}(\mathrm{gcbg}).\mathrm{instrs}] \,\}
\end{aligned}
$$

Figure 5. Abstract Specification of the Memory Allocator

mdisj uses the undef option for reg, stk, tab, and shp to ensure that if a memory is split into disjoint pieces, each of these indivisible components can only appear in one of the pieces.

Possible Worlds The model of possible worlds that we use to implement WorldSpec is based very closely on Dreyer *et al.* [14]. For space reasons, we will therefore not present the details of the model in this paper and instead refer the reader to the appendix [15]. (See Section 8 for a more detailed discussion of how our model relates to Dreyer *et al.*'s.)

That said, there are a few salient aspects of the model that are needed for understanding the definition of compositional program equivalence that we will give in Section 7. In the model, worlds W are 3-tuples (k, ω, GR), where: k is W's step level, ω is a finite set of state transition systems (STS's) of the sort described in Section 3, and GR is a global invariant governing the entire memory. We call the STS's *islands* because they govern disjoint pieces of memory. Two memories are related by W if (1) they satisfy its global invariant GR, and (2) they can be split into disjoint memories (one for each island) such that the n-th pair of memories satisfies the "local" memory relation determined by the current state of the n-th island in ω. In our definition of program equivalence, we will use GR to enforce the property that all reachable data is live. We need to use a global invariant since reachability cannot be determined by looking at a local subheap. We use islands to express all other assumptions about memory.

6. Assumptions About the Memory Allocator

Figure 5 shows the assumptions we make about memory allocation and garbage collection, in the form of the specification AllocSpec. Given as input a starting physical address gcbg, the runtime system represented by $\mathcal{A}$ will return a 4-tuple $(\mathrm{init}, \mathrm{alloc}, \mathrm{instrs}, I)$, where: init is the starting address of the initialization routine that sets up the runtime system, alloc is the starting address of the allocator, instrs is the list of instructions defining the runtime system, which are assumed to be loaded at address gcbg, and I is a private invariant of the runtime system, which describes when a logical memory's lookup table is in sync with its system heap. The assumption about instrs is joined together with the private invariant I to form the memory predicate MR defined at the bottom of the figure.

Assuming init is invoked with the runtime system code in the right place, and with a return address placed in wk_4, its specification says it will return control in a memory that satisfies $MR(\mathrm{gcbg})$, along with the global invariant $GR(\mathrm{gcbg})$ that all reachable data is live.

Assuming that alloc is invoked in a physical memory Φ represented abstractly by the logical memory $\mathbf{M}$, that $\mathbf{M}$ satisfies the MR and GR properties, that the number of cells to be allocated (n) is stored in wk_5, and that the return address is stored in wk_4, the specification of alloc says that it will return a pointer to a fresh n-word block in wk_5, and that the memory it returns ($\mathbf{M}'$) will continue to satisfy all the aforementioned invariants. Moreover, while the lookup table and system heap of $\mathbf{M}'$ may be completely different from those of $\mathbf{M}$, the contents of $\mathbf{M}$ must remain otherwise unchanged. This does not of course prevent the allocator from having performed a GC: any logical pointer that was not reachable in $\mathbf{M}$ before the call to alloc may very well be marked as dead in the lookup table of the post-allocation $\mathbf{M}'$, but any pointer that was reachable in $\mathbf{M}$ will still be reachable in $\mathbf{M}'$ and thus, by the definition of GR, still be live.

Our specification of the runtime system provides considerable flexibility—for example, it should be satisfied by either a mark-and-sweep or a copying collector because the specification says nothing about the private invariant of the runtime system. However, it *does* assume that the collector places no restrictions (such as read or write barriers) on what the mutator does to live data. We believe it should be possible to adapt our approach to a wider range of collectors, but we leave that to future work.

7. Compositional Program Equivalence

The logical relation $\mathcal{E}[\![\tau]\!]\rho$ characterizes what it means for two *computations* to be logically equivalent, but ultimately what we really care about is whether a pair of **HIGH** and **LOW** *programs* are logically equivalent. What, one may wonder, is the difference between computations and programs? In short, a program is what you *write*, and a computation is what you *run*. That is, a program is a piece of relocatable code that must be linked with other programs and loaded into memory before it can be executed, whereas a computation describes the "next" thing to be executed in a running machine configuration. For the **HIGH** language, the distinction between computations and programs can be easily glossed over because the operational semantics of **HIGH** is defined directly on **HIGH** programs. For the **LOW** language, however, especially given the ability to write self-modifying code, it is important to distinguish the two notions. In this section, we explain what a **LOW** program is and how to define logical equivalence between **HIGH** and **LOW** programs, and we then present our key technical results.

7.1 Equivalence of HIGH and LOW Programs

As can already be seen from our motivating example in Section 3.1, we define a **LOW** program p to be a function from two code pointers to a list of instructions. The first of the code pointer inputs is assumed to be the address of the memory allocation routine, and the second is assumed to be the address where the list of

$$
\begin{array}{l}
\mathcal{H}.\text{Prog} \overset{\text{def}}{=} \{\, e \mid \text{floc}(e) = \emptyset \,\} \\
\mathcal{L}.\text{Prog} \overset{\text{def}}{=} \{\, p \in \text{PAddr} \times \text{PAddr} \to \text{List Instruction} \,\} \\[1ex]
\quad \mathcal{D}[\![\cdot]\!] \overset{\text{def}}{=} \emptyset \\
\mathcal{D}[\![\Delta, \alpha]\!] \overset{\text{def}}{=} \{\, (\rho, \alpha \mapsto R) \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge R \in \text{TyValRel} \,\} \\[1ex]
\qquad \mathcal{G}[\![\cdot]\!]\rho \overset{\text{def}}{=} \{\, (W, \mathbf{v}, \emptyset) \mid W \in \text{World} \wedge \mathbf{v} \in \mathcal{L}.\text{Val} \,\} \\
\mathcal{G}[\![\Gamma, x : \tau]\!]\rho \overset{\text{def}}{=} \{\, (W, \mathbf{v}, (\gamma, x \mapsto v)) \mid \exists \mathbf{v}_1, \mathbf{v}_2. \\
\qquad\qquad (W, \mathbf{v}, \langle\rangle) \in \Box\overline{(\mathcal{L}.\text{pair}(\mathbf{v}_1, \mathbf{v}_2), \mathcal{H}.\text{Val} \times \mathcal{H}.\text{Mem})} \wedge \\
\qquad\qquad (W, \mathbf{v}_1, v) \in \mathcal{V}[\![\tau]\!]\rho \wedge (W, \mathbf{v}_2, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho \,\} \\[1ex]
W_k^\circ(\mathcal{A}, \text{gcbg}) \overset{\text{def}}{=} (k, [\iota^{\text{regstk}}, \iota^{\text{htyping}}, \iota^{\text{gc}}(\mathcal{A}, \text{gcbg})], GR^\circ(\mathcal{A}, \text{gcbg})) \\[1ex]
\Delta; \Gamma \vdash \text{bg} \approx_W e : \tau \;\overset{\text{def}}{=} \\
\;\forall W' \sqsupseteq W.\, \forall \rho \in \mathcal{D}[\![\Delta]\!]\,.\, \forall (\mathbf{v}, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho(W'). \\
\quad ((\underline{\text{bg}}, \lfloor \text{wk}_0 \rfloor, \lfloor \text{wk}_5 \rfloor, \{\, \mathbf{M} \mid \mathbf{M}.\text{reg}(\text{sv}_0) = \mathbf{v} \,\}), \gamma\rho e) \in \mathcal{E}[\![\tau]\!]\rho(W') \\
\;\text{where } \gamma\rho e ::= e[\rho(\alpha).\tau_2/\alpha][\gamma(x)/x] \\[1ex]
\Delta; \Gamma \vdash p \approx e : \tau \;\overset{\text{def}}{=} \\
\;\emptyset; \Delta; \Gamma \vdash e : \tau \wedge \\
\;\forall \mathcal{A}, \text{gcbg}, \text{bg}.\, \forall k, W \sqsupseteq W_k^\circ(\mathcal{A}, \text{gcbg}).\, \forall (\mathbf{M}, M) \in \mathcal{M}(W). \\
\;\forall \mathbf{M}'.\, \mathbf{M}' = \mathbf{M} \uplus \{[\text{bg} \Mapsto p(\mathcal{A}(\text{gcbg}).\text{alloc}, \text{bg})]\}_{\text{code}} \implies \\
\;\exists W' \sqsupseteq W.\, \text{lev}(W') = \text{lev}(W) \wedge (\mathbf{M}', M) \in \mathcal{M}(W') \wedge \\
\qquad\qquad \Delta; \Gamma \vdash \text{bg} \approx_{W'} e : \tau
\end{array}
$$

Figure 6. Program Equivalence

instructions returned by the program will be loaded into memory. When listing the code of a program (*e.g.,* in Figure 2), we write line numbers on selected lines of code (*e.g.,* bg + 3) to indicate the physical addresses where we expect the code to be loaded, but note that (1) these addresses are always relative to the second parameter of the program (typically named bg), so that the code is always relocatable, and (2) the notation is merely suggestive—the line numbers are not part of the actual program.

Now, concerning equivalence of **HIGH** and **LOW** programs: it is possible to define a notion of logical equivalence strictly between *closed* programs, but we will find it useful when reasoning about compiler correctness to generalize this relation to one on *open* programs. On the **HIGH** side, an open program is simply an expression e with free variables (and no free locations), but what is an open program on the **LOW** side? In order to answer this, we need to pick an environment-passing convention, specifying what is an official low-level representation of a high-level environment and where the low-level program expects to get its environment data. In Figure 6, we give a logical relation $\mathcal{G}[\![\Gamma]\!]\rho$ between low-level values and high-level environments. This relation specifies that a high-level environment γ should be represented as a linked list of low-level values that are componentwise related to the high-level values in the range of γ. We assume that the environment data is passed in the register sv_0—this assumption will be codified in the definition of the program equivalence relation $\approx$ (see below).

Before we define program equivalence, we must also specify the invariants on memories that are required for executing programs. These conditions are represented by the *initial* worlds W_k° in Figure 6. (The k in W_k° simply determines the step level but does not otherwise affect the definition.) The initial worlds consist of three islands and one global invariant: ι^{regstk} owns the register file and the stack in the **LOW** memory and requires that callee-save registers and stack be preserved before and after function calls (this is accomplished using a combination of private and public transitions as discussed in Section 3.5); ι^{htyping} requires that the heap typing in the **HIGH** memory should only grow in future worlds; $\iota^{\text{gc}}(\mathcal{A}, \text{gcbg})$ owns the lookup table and the system heap and enforces the private $\mathcal{A}.MR(\text{gcbg})$ invariant of the runtime system (Figure 5); and the global invariant $GR^\circ(\mathcal{A}, \text{gcbg})$ specifies that all reachable blocks in the **LOW** memory are live ($\mathcal{A}.GR(\text{gcbg})$ in Figure 5) and that the **HIGH** memory satisfies its heap typing. All these components of W_k° are formally defined in the appendix [15].

We are now ready to define the program equivalence relation $\approx$. The judgment $\Delta; \Gamma \vdash p \approx e : \tau$ says that a **LOW** program p and a **HIGH** program e are equivalent if the following is true. First, let k be any starting step level, and let W be any future world of W_k° that does not already impose any invariants on the code segment of p. (This latter condition is guaranteed by the assumption that we are given initial memories $\mathbf{M}$ and M related by W, but where $\mathbf{M}$ does *not* contain the code segment of p. We use the notation $\mathbf{M}_1 \uplus \mathbf{M}_2$ here to denote the "smallest" memory in $\text{mdisj}(\mathbf{M}_1, \mathbf{M}_2)$, in a sense defined formally in the appendix.)

Under these assumptions, we must be able to:

1. Extend W to a future world W' (of the same step level) such that W' relates $\mathbf{M}'$ and M, where $\mathbf{M}'$ is $\mathbf{M}$ extended with the code of p. Intuitively, this step affords p the opportunity to "own" its own code segment by extending W with an island governing it. Typically, this island will take the form of an invariant stating that p's code must never be modified, although in the case of self-modifying code we would instead define the island to be a state transition system (as described in Section 3.4).
2. Show that the **HIGH** *computation* e is related (under the world W' constructed in the previous step) to the **LOW** *computation* $\mathbf{e}$ that starts at the beginning (bg) of p's code segment. This subgoal is encapsulated in the *open computation equivalence* judgment $\Delta; \Gamma \vdash \text{bg} \approx_{W'} e : \tau$ (Figure 6). This judgment says that, for any future world W'' of W', for any relational interpretation ρ of the type variables in Δ, and for any environments $\mathbf{v}$ and γ related under W'' by $\mathcal{G}[\![\Gamma]\!]\rho$, the **HIGH** computation $\gamma\rho e$ is logically related (by $\mathcal{E}[\![\tau]\!]\rho$, under W'') to the **LOW** computation starting at address bg. The other three components of the **LOW** computation stipulate, respectively, that (1) the computation expects to find its return address stored in wk_0, (2) the computation will store its resulting value in wk_5, and (3) the computation expects to find its environment stored in sv_0. One can think of this last assumption about sv_0 as having the effect of "closing" p, in much the same way that γ and ρ close e.

7.2 Compiler Correctness and Other Technical Results

Our first result is an *adequacy* theorem for our program equivalence relation $\approx$. The statement of the theorem refers to the following simple loader for the **LOW** language: $\text{load}(\mathcal{A}, p)$ first runs the initialization routine of the memory allocator $\mathcal{A}$, then executes p, and finally halts as soon as it gets control back from p:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{load}(\mathcal{A}, p) ::= \text{let } (\text{init}, \text{alloc}, \text{gcinstrs}, \_) := \mathcal{A}(105),} \\
\multicolumn{4}{l}{\qquad\qquad\quad \text{instrs}^p := p(\text{alloc}, 105 + |\text{gcinstrs}|),} \\
\multicolumn{4}{l}{\qquad\qquad\quad \text{loadinstrs}^p := [} \\
(*\,100\,*) & \texttt{move} & \lfloor \text{wk}_4 \rfloor & \underline{102} \\
 & \texttt{jmp} & \underline{\text{init}} & \\
(*\,102\,*) & \texttt{move} & \lfloor \text{wk}_0 \rfloor & \underline{104} \\
 & \texttt{jmp} & \underline{105 + |\text{gcinstrs}|} & \\
(*\,104\,*) & \texttt{halt} & & \\
\multicolumn{4}{l}{] +\!\!+ \text{gcinstrs} +\!\!+ \text{instrs}^p \quad \text{in}} \\
\multicolumn{4}{l}{\{\, (\Phi, 100) \in \text{PConf} \mid \Phi.\text{code} \supseteq [100 \Mapsto \text{loadinstrs}^p] \,\}}
\end{array}
$$

The adequacy theorem states that closed **HIGH** and **LOW** programs that are equivalent according to $\approx$ must equi-terminate when loaded by the above loader.

Theorem 1 (Adequacy)**.** For all $\emptyset; \emptyset \vdash p \approx e : \tau$,
$\forall \mathcal{A} \in \text{AllocSpec}.\, \forall (\Phi, \text{pc}) \in \text{load}(\mathcal{A}, p).\, \forall h.$
both (Φ, pc) and (h, e) diverge or both halt without fail.

One might think that this adequacy result is weak because the loader does not inspect the result returned by the program p. How-

```
$\mathrm{Papp}(p_1, p_2)$ ::= $\lambda$ alloc, bg.
let $\mathrm{instrs}_1 := p_1(\mathrm{alloc}, \mathrm{bg}+4)$, $c_1 := |\mathrm{instrs}_1|$,
   $\mathrm{instrs}_2 := p_2(\mathrm{alloc}, \mathrm{bg}+c_1+6)$, $c_2 := |\mathrm{instrs}_2|$   in   [
  bg                  plus    $\lfloor sp \rfloor$          $\lfloor sp \rfloor$                  $\underline{2}$
                      move    $\langle sp-2\rangle_s$      $\lfloor wk_0 \rfloor$
                      move    $\langle sp-1\rangle_s$      $\underline{0}$
                      move    $\lfloor wk_0 \rfloor$        $\underline{bg+c_1+4}$
                        $\mathrm{instrs}_1$
  $bg+c_1+4$          move    $\langle sp-1\rangle_s$      $\lfloor wk_5 \rfloor$
                      move    $\lfloor wk_0 \rfloor$        $\underline{bg+c_1+c_2+6}$
                        $\mathrm{instrs}_2$
  $bg+c_1+c_2+6$      move    $\lfloor wk_0 \rfloor$        $\langle sp-2\rangle_s$
                      move    $\lfloor wk_1 \rfloor$        $\langle sp-1\rangle_s$
                      move    $\lfloor wk_2 \rfloor$        $\lfloor wk_5 \rfloor$
                      minus   $\lfloor sp \rfloor$          $\lfloor sp \rfloor$                  $\underline{2}$
                      jmp     $\langle wk_1+0\rangle_h$
]
```

Figure 7. Compilation of Function Application

ever, together with the compositionality result below, one can link the program p with arbitrary well-behaved "test" programs and the linked programs are guaranteed to behave the same.

In order to show that our logical relations are sufficiently populated—an important desideratum, as we explained in the introduction—we have written a very naïve compiler from **HIGH** to **LOW** and proved that every source program is related to its compiled low-level program by our program equivalence. Specifically, we have implemented a low-level construct corresponding to each high-level construct and then defined the compiler purely inductively on the structure of source programs using these low-level constructs. For each construct, we have shown a corresponding "compatibility" lemma (following Pitts' terminology [21]), meaning that program equivalence is preserved under said construct. This implies that equivalent programs behave the same under arbitrary well-behaved contexts.

One example of a high-level construct is function application. Figure 7 shows its low-level realization as a simple linking program $\mathrm{Papp}(p_1, p_2)$. We assume here that p_1 is some program computing a value of function type $\tau' \to \tau$, and p_2 is some program computing a value of the argument type τ'.

The program begins by bumping up the stack pointer twice, pushing the return address (stored in wk_0) onto the first new stack slot, and clearing the second one with 0. The clearing instruction (at bg + 2) is needed because the stack slot at $\langle \mathrm{sp}-1\rangle_s$ might otherwise contain a dangling pointer; thus, in order to maintain the global invariant that all reachable data is live, we must clear $\langle \mathrm{sp}-1\rangle_s$ before passing control to p_1. When p_1 returns, we assume it returns a pointer to a function closure in wk_5, we store that pointer in the cleared stack slot, and we proceed to execute p_2. After p_2 returns, we move p_1's closure pointer into wk_1, the argument value (returned by p_2) into wk_2, and the original return address of $\mathrm{Papp}(p_1, p_2)$ into wk_0. We then pop off two stack slots and make a tail call to the code pointer stored in wk_1's closure.

The compatibility result for Papp can be seen as a compositionality result for our program equivalence relation.

Lemma 1 (Compatibility: App)**.**

$$\Delta;\Gamma \vdash p_1 \approx e_1 : \tau' \to \tau \wedge \Delta;\Gamma \vdash p_2 \approx e_2 : \tau' \implies \\ \Delta;\Gamma \vdash \mathrm{Papp}(p_1, p_2) \approx e_1\ e_2 : \tau$$

Another related construct of course is λ-abstraction. Figure 8 shows its low-level realization as the program $\mathrm{Pabs}(p)$. We assume here that p is a program implementing the body of a λ-abstraction, under the assumption that the argument of that abstraction is the first element in the linked list environment stored at sv_0.

```
$\mathrm{Pabs}(p)$ ::= $\lambda$ alloc, bg.
let $\mathrm{code} := p(\mathrm{alloc}, \mathrm{bg}+16)$, $c := |\mathrm{code}|$    in   [
  bg            move    $\lfloor wk_4 \rfloor$        $\underline{bg+3}$
                move    $\lfloor wk_5 \rfloor$        $\underline{2}$
                jmp     $\underline{alloc}$
  $bg+3$        move    $\langle wk_5+0\rangle_h$    $\underline{bg+6}$
                move    $\langle wk_5+1\rangle_h$    $\lfloor sv_0 \rfloor$
                jmp     $\lfloor wk_0 \rfloor$
  $bg+6$        plus    $\lfloor sp \rfloor$          $\lfloor sp \rfloor$             $\underline{2}$
                move    $\langle sp-2\rangle_s$      $\lfloor wk_0 \rfloor$
                move    $\langle sp-1\rangle_s$      $\lfloor sv_0 \rfloor$
                move    $\lfloor wk_4 \rfloor$        $\underline{bg+12}$
                move    $\lfloor wk_5 \rfloor$        $\underline{2}$
                jmp     $\underline{alloc}$
  $bg+12$       move    $\langle wk_5+0\rangle_h$    $\lfloor wk_2 \rfloor$
                move    $\langle wk_5+1\rangle_h$    $\langle wk_1+1\rangle_h$
                move    $\lfloor sv_0 \rfloor$        $\lfloor wk_5 \rfloor$
                move    $\lfloor wk_0 \rfloor$        $\underline{bg+c+16}$
    code
  $bg+c+16$     move    $\lfloor sv_0 \rfloor$        $\langle sp-1\rangle_s$
                minus   $\lfloor sp \rfloor$          $\lfloor sp \rfloor$             $\underline{2}$
                jmp     $\langle sp-0\rangle_s$
]
```

Figure 8. Compilation of λ-Abstraction

The actual computation of the program is quite simple, because it merely allocates a closure representing the λ-abstraction and returns it (just like the example in Section 3.1). The closure consists of a code pointer (to bg + 6) and an environment pointer, which is of course just whatever environment was passed to $\mathrm{Pabs}(p)$ in sv_0. Whenever the closure is invoked (by jumping to bg + 6), it first saves the return address (stored in wk_0), as well as the callee-save register sv_0, by pushing them on the stack. It then pushes the argument value $\lfloor \mathrm{wk}_2 \rfloor$ onto the front of the environment linked-list headed by $\langle \mathrm{wk}_1+1\rangle_h$ (which requires allocating two new memory cells), and stores a pointer to this extended environment in sv_0 before executing p. Finally, when p returns, it pops the stack twice, and restores the original contents of the callee-save sv_0, before jumping to the return address that was stored on the stack.

The compatibility result for Pabs is as follows:

Lemma 2 (Compatibility: Abs)**.**

$$\Delta;\Gamma, x : \tau' \vdash p \approx e : \tau \implies \\ \Delta;\Gamma \vdash \mathrm{Pabs}(p) \approx \lambda x{:}\tau'.\, e : \tau' \to \tau$$

Using Papp, Pabs, and all the other **LOW** program constructors, we can define a compiler $(\!|\Gamma \vdash e|\!)$ in a simple, syntax-directed fashion, *e.g.,*

$$(\!|\Gamma \vdash e_1\ e_2|\!) ::= \mathrm{Papp}((\!|\Gamma \vdash e_1|\!), (\!|\Gamma \vdash e_2|\!))$$
$$(\!|\Gamma \vdash \lambda x{:}\tau.\, e|\!) ::= \mathrm{Pabs}((\!|\Gamma, x{:}\tau \vdash e|\!))$$

and then establish the following compiler correctness result:

Theorem 2 (Compiler Correctness)**.** For $\emptyset;\Delta;\Gamma \vdash e : \tau$,

$$\Delta;\Gamma \vdash (\!|\Gamma \vdash e|\!) \approx e : \tau$$

The theorem is easily provable by induction on e, using the appropriate compatibility lemma in each case.

Finally, we prove the self-modifying awkward program p_{awk} is equivalent to the high-level awkward program e_{awk} from Figure 2.

Theorem 3. $\emptyset;\emptyset \vdash p_{\mathrm{awk}} \approx e_{\mathrm{awk}} : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int}$

As a corollary, we can see that for any $\emptyset;\emptyset;\emptyset \vdash e : \mathsf{unit} \to \mathsf{unit}$, both $e_{\mathrm{awk}}\ \ e$ and $\mathrm{load}(\mathcal{A}, \mathrm{Papp}(p_{\mathrm{awk}}, (\!|\emptyset \vdash e|\!)))$ equi-terminate. Similarly, for any $\emptyset;\emptyset;\emptyset \vdash e : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int}) \to \tau$, both $e\ e_{\mathrm{awk}}$ and $\mathrm{load}(\mathcal{A}, \mathrm{Papp}((\!|\emptyset \vdash e|\!), p_{\mathrm{awk}}))$ equi-terminate.

Detailed proofs of all these results appear in the companion technical appendix [15].

8. Related and Future Work

There is a huge body of work on compiler correctness and semantics for low-level code. We focus on the most closely related work.

Compositional Compiler Correctness As explained in the introduction, the overall motivation of our work is very similar to that of Benton and Hur [5], and our use of logical relations to build an extensional, compositional notion of equivalence between high- and low-level languages is inspired directly by their work. However, there are significant differences between our work and theirs.

First, they define a relation between a purely functional PCF-like language and an SECD machine, whereas we relate a more expressive, impure, ML-like high-level language to an assembly language that is significantly more low-level and realistic than SECD. Reasoning about compositional equivalence in our setting is significantly more complex, not least because we must deal with reasoning about the heap and the presence of a garbage collector. We make essential use of Kripke logical relations for this purpose.

That said, there is a sense in which our setting makes the problem easier. One of Benton and Hur's goals was to develop a model of low-level programs that would admit program equivalences (such as commutativity of addition) whose validity depend on the purely functional nature of the source language. Toward this end, they related low-level programs to *denotations* of high-level programs, so that one could use domain-theoretic reasoning to establish the purely functional equivalences of interest. These half-operational, half-denotational relations were of necessity asymmetric. In particular, they employ biorthogonality—but only on the low-level side of the relation—as well as step-indexing—but only in defining one direction of approximation (in the other direction, they use an admissible closure operation). As a result, proving in their setting that a high- and low-level program are *equivalent* really involves doing two proofs (one for each direction of approximation) using very different technical machinery.

In our work, we sidestep this problem because the ML-like lack of effect encapsulation in our high-level language causes it to have a relatively weak equational theory that simply does not admit the kinds of purely functional equivalences that Benton and Hur were interested in. Nevertheless, as our motivating example illustrates, there are still plenty of interesting equivalences in our setting, particularly involving uses of local state. Moreover, our logical relations, being entirely operational and defined in language-generic fashion, are inherently symmetric, making them easier to use.

More recently, Benton and Hur [6] have generalized their technique to a compiler for a polymorphic (yet still purely functional) language, but their logical relations are still asymmetric.

Chlipala [11] proposes a syntactic approach to proving compositional compiler correctness. His idea is to establish a set of criteria for high-low compilation relations, such that the results of different compilers can be correctly linked so long as their compilation relations satisfy these criteria. However, the criteria are still syntactic, and thus he cannot reason for instance about our motivating example, wherein the high-low equivalence depends on semantic reasoning about local state. Moreover, Chlipala only considers a fairly high-level target language that is a CPS version of the source.

Jaber and Tabareau [16] propose an alternative approach to compositional compiler correctness based on type preservation. Instead of proving compiler correctness directly, they prove that the compiler is type-preserving, but their source language has such a rich type system (with dependent refinement types) that this effectively implies correctness. Like Benton and Hur, they compile a purely functional language to an SECD machine, but their correctness result only applies to terminating programs.

None of the aforementioned work considers a target language that supports garbage collection.

Kripke Logical Relations Our logical relation is based closely on Dreyer, Neis and Birkedal's (hereafter, DNB) [14], which was in turn a refinement and generalization of Ahmed, Dreyer and Rossberg's [2]. (Of course, these are but the latest in a long line of work on Kripke logical relations spanning decades—see *loc. cit.*, as well as Pitts and Stark [22], for further pointers to the literature.)

DNB's main goal was to show how a Kripke model based on state transition systems could be extended in orthogonal ways to exploit the *absence* of certain features, namely higher-order state and/or control effects. For the **HIGH** language considered here, in which there are no control operators, DNB showed how to extend their baseline model with *private transitions* (Section 3.5) and demonstrated the utility of private transitions in reasoning about a variety of challenging contextual equivalences involving local state. For our purposes, private transitions have proved useful in formalizing the "well-bracketing" assumptions about the stack and callee-save registers, assumptions which indeed rely on the absence of control operators. Extending the **HIGH** language with control operators would thus necessitate a significant change to our Kripke model, precisely because the compilation strategy for **HIGH** would need to change as well. We leave this problem to future work.

Despite the close connection, our logical relation diverges from DNB's in several ways. First and foremost, whereas DNB's relation was only designed to reason about high-level programs, the whole point of our model is to allow us to relate high- and low-level programs. As a result, we have no "fundamental theorem of logical relations" because one cannot even state such a theorem for a relation between two languages. Instead, we prove a *compiler correctness* result, whose proof mirrors that of the usual fundamental theorem. Our consideration of low-level programs has also led us to make a clear distinction between *programs* and *computations*, and between the notions of equivalence thereon. This seems to us an interesting and important distinction that is worth investigating further.

Dealing with low-level programs introduces significant technical complexity. In order to isolate this complexity, we factor the presentation of our relation generically w.r.t. a language specification "interface" $\mathrm{LangSpec}$. This helps to clarify the structure of our Kripke logical relation, bringing into relief its essentially symmetric high-level structure, as well as the components of the model that are language-dependent (namely the two implementations of $\mathrm{LangSpec}$, where most of the complexity lies). Although we have in this paper only instantiated our model so as to relate **HIGH** and **LOW** programs, one may also instantiate the model to relate **HIGH** and **HIGH** programs, or **LOW** and **LOW** programs. The former model would be largely similar to Dreyer *et al.*'s; the latter would enable one to reason about equivalence of low-level programs directly, which may prove useful in reasoning about correctness of low-level optimizations, although this remains to be explored.

As far as possible worlds are concerned, although ours are largely similar to DNB's, we have extended their worlds in one relatively straightforward way: in addition to *local* invariants (expressed in our possible worlds by *islands*), our worlds also permit one to express a *global* invariant. We exploit this added functionality to encode the allocator's invariant that all reachable data are live (Sections 3.6 and 6).

Lastly, as far as the high-level structure of the logical relation is concerned, ours is also quite similar to DNB's, except that our interpretation of reference types is somewhat different. Various approaches to interpreting reference types have been proposed in the literature; our present interpretation is in a more "extensional" style than either DNB's or Ahmed *et al.*'s [2], in the sense that it avoids dependence on too many details of how possible worlds are structured. This enables us to present it, as we have, in a more "world-generic" fashion. Among existing accounts, our present formulation is fairly close to the denotational one given by Birkedal, Støvring and Thamsborg [7], but the jury is still out on which of these formulations is most felicitous.

Although the primary benefit of presenting our logical relation language- and world-generically is to clarify its (admittedly complex) structure, it also enables us to prove a few "structural" lemmas generically as well, most notably *monotonicity*. (That we can prove monotonicity generically should not be surprising, given that it is essentially baked into the logical relation via the quantification over future worlds in certain cases and the use of the $\Box$ operator in others.) Most of the interesting theorems, however, cannot be stated, let alone proven, without talking about the details of the $\mathrm{LangSpec}$'s for **HIGH** and **LOW**.

Garbage Collection Torp-Smith *et al.* [24] prove the correctness of a Cheney copying collector using separation logic. They specify the behavior of a correct garbage collector in terms of an isomorphism between the reachable portions of the initial and final heaps. Our specification is broadly similar in that we express the reachable portion of the heap in our logical memories. We construct our entire $\mathrm{LangSpec}$ for **LOW** around these logical memories.

Building on the work of Torp-Smith *et al.*, McCreight *et al.* [20] develop a garbage collector interface that is general enough to characterize a variety of different collectors, including incremental copying collectors with read and write barriers. They prove in Coq that various collectors implement this interface, and that various mutator programs respect it. In more recent work, McCreight *et al.* [19] extend Leroy's Compcert compiler [18] with support for garbage collection, by building the mutator-collector interface into the design of a new intermediate language, GCminor. They prove semantics preservation (mostly) for a compiler from a purely functional, typed language—Dminor—to GCminor, but (as in Compcert) not compositional correctness.

Self-Modifying Code Cai *et al.* [9] provide one of the only formal accounts of how to verify self-modifying code. They use Hoare-style separation-based reasoning to enable local reasoning about modifications to code, in much the same way that standard separation logic enables local reasoning about the heap. We adopt a similar approach, using possible worlds to impose local invariants on—and, more generally, to establish local state transition systems governing—both the heap and the code segment. Ours is the first relational model for reasoning about (compositional) equivalence of self-modifying programs.

Future Work Given our ability to reason about self-modifying code, one direction for future work is to adapt this functionality to reason about more practical applications, such as a just-in-time compiler or a dynamic linker/loader.

We have shown compositionality of our high-low relation by showing that it is closed under the linking constructs expressible at the level of our **HIGH** language, such as function application. In future work, we would like to prove that our relation is also compositional w.r.t. a more realistic linking language.

The concrete logical relation we have presented here assumes a uniform data representation. It is possible in principle to define the meaning of language forms like $\mathrm{pair}(\mathbf{v}_1, \mathbf{v}_2)$ in a non-uniform way—*e.g.,* to enable flattening—but it is not currently possible in our language-generic framework to define such language forms in a *type*-specialized manner. We leave a serious examination of this issue to future work.

Lastly, it is unclear how to scale our techniques to reason about compositional correctness of a *multi-phase* compiler because the step-indexed logical relations we define are not obviously transitive. This is a well-known problem with step-indexed logical relations [1], and it seems a fresh idea is needed to circumvent it.

Acknowledgments

We would like to thank Jacob Thamsborg for pointing out a very subtle technical oversight in an earlier draft of this paper.

References

[1] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[2] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[3] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

[4] A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.

[5] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[6] N. Benton and C.-K. Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, Apr. 2010.

[7] L. Birkedal, K. Støvring, and J. Thamsborg. A relational realizability model for higher-order stateful ADTs. Submitted for publication, 2010.

[8] N. Bohr. *Advances in Reasoning Principles for Contextual Equivalence and Termination*. PhD thesis, IT University of Copenhagen, 2007.

[9] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *PLDI*, 2007.

[10] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI*, 2007.

[11] A. Chlipala. Syntactic proofs of compositional compiler correctness, 2009. Submitted for publication.

[12] A. Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010.

[13] Z. Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Denis Diderot, July 2009.

[14] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, 2010.

[15] C.-K. Hur and D. Dreyer. Technical appendix for this paper, 2010. URL: `http://www.mpi-sws.org/~dreyer/papers/lrmlasm/`.

[16] G. Jaber and N. Tabareau. Krivine realizability for compiler correctness. In *LOLA*, 2010.

[17] J.-L. Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68:53–78, 1994.

[18] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[19] A. McCreight, T. Chevalier, and A. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*, 2010.

[20] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *PLDI*, 2007.

[21] A. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7. MIT Press, 2005.

[22] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[23] E. Sumii. A complete characterization of observational equivalence in polymorphic lambda-calculus with general references. In *CSL*, 2009.

[24] N. Torp-Smith, L. Birkedal, and J. C. Reynolds. Local reasoning about a copying garbage collector. *TOPLAS*, 30(4), 2008.