The Impact of Higher-Order State and Control Effects on Local Relational Reasoning

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Georg Neis
MPI-SWS
neis@mpi-sws.org

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Abstract

Reasoning about program equivalence is one of the oldest problems in semantics. In recent years, useful techniques have been developed, based on bisimulations and logical relations, for reasoning about equivalence in the setting of increasingly realistic languages—languages nearly as complex as ML or Haskell. Much of the recent work in this direction has considered the interesting representation independence principles *enabled* by the use of local state, but it is also important to understand the principles that powerful features like higher-order state and control effects *disable*. This latter topic has been broached extensively within the framework of game semantics, resulting in what Abramsky dubbed the "semantic cube": fully abstract game-semantic characterizations of various axes in the design space of ML-like languages. But when it comes to reasoning about many actual examples, game semantics does not yet supply a useful technique for proving equivalences.

In this paper, we marry the aspirations of the semantic cube to the powerful proof method of *step-indexed Kripke logical relations*. Building on recent work of Ahmed, Dreyer, and Rossberg, we define the first fully abstract logical relation for an ML-like language with recursive types, abstract types, general references and call/cc. We then show how, under orthogonal restrictions to the expressive power of our language—namely, the restriction to first-order state and/or the removal of call/cc—we can enhance the proving power of our possible-worlds model in correspondingly orthogonal ways, and we demonstrate this proving power on a range of interesting examples. Central to our story is the use of *state transition systems* to model the way in which properties of local state evolve over time.

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, observational equivalence, higher-order state, local state, first-class continuations, exceptions, state transition systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-60558-794-3/10/09. . . $10.00

1. Introduction

Reasoning about program equivalence is one of the oldest problems in semantics, with applications to program verification ("Is an optimized program equivalent to some reference implementation?"), compiler correctness ("Does a program transformation preserve the semantics of the source program?"), representation independence ("Can we modify the internal representation of an abstract data type without affecting the behavior of clients?"), and more besides.

The canonical notion of program equivalence for many applications is *observational* (or *contextual*) equivalence. Two programs are observationally equivalent if no program context can distinguish them by getting them to exhibit observably different input/output behavior. Reasoning about observational equivalence directly is difficult, due to the universal quantification over program contexts. Consequently, there has been a huge amount of work on developing useful models and logics for observational equivalence, and in recent years this line of work has scaled to handle increasingly realistic languages—languages nearly as complex as ML or Haskell, with features like general recursive types, general (higher-order) mutable references, and first-class continuations.

The focus of much of this recent work—*e.g.,* environmental bisimulations [36, 17, 32, 35], normal form bisimulations [34, 16], step-indexed Kripke logical relations [4, 2, 3]—has been on establishing some effective techniques for reasoning about programs that actually *use* the interesting, semantically complex features (state, continuations, etc.) of the languages being modeled. For instance, most of the work on languages with state concerns the various kinds of representation independence principles that arise due to the use of *local state* as an abstraction mechanism.

But of course this is only part of the story. When features are added to a language, they also enrich the expressive power of program *contexts*. Hence, programs that do *not* use those new features, and that are observationally equivalent in the absence of those features, might not be observationally equivalent in their presence. One well-known example of this is the loss of referential transparency in an impure language like ML. Another shows up in the work of Johann and Voigtländer [15], who study the negative impact that Haskell's strictness operator `seq` has on the validity of short-cut fusion and other free-theorems-based program transformations. In our case, we are interested in relational reasoning about *stateful* programs, so we will be taking a language with some form of mutable state as our baseline. Nonetheless, we feel it is important not only to study the kinds of local reasoning principles that stateful programming can *enable*, but also to understand the principles that powerful features like higher-order state and control effects *disable*.

This latter topic has been broached extensively within the framework of *game semantics*. In the 1990s, Abramsky set forth a research programme (subsequently undertaken by a number of people) concerning what he called the *semantic cube* [19, 1, 24].

The idea was to develop fully abstract game-semantic characterizations of various axes in the design space of ML-like languages. For instance, the absence of mutable state can be modeled by restricting game strategies to be *innocent*, and the absence of control operators can be modeled by restricting game strategies to be *well-bracketed*. These restrictions are orthogonal to one another and can be composed to form fully abstract models of languages with different combinations of effects. Unfortunately, when it comes to reasoning about many actual examples, these game-semantics models do not yet supply a useful technique for proving programs equivalent, except in fairly restricted languages.

One possible reason for the comparative lack of attention paid to this issue in the setting of relational reasoning is that some key techniques that have been developed for reasoning about local state—notably, Pitts and Stark's method of *local invariants* [28]—turn out to work just as well in a language with higher-order state and call/cc as they do in the simpler setting (first-order state, no control operators) in which they were originally proposed. Before one can observe the negative impact of certain language features on relational reasoning principles, one must first develop a proof technique that actually *exploits* the absence of those features!

1.1 Overview

In this paper, we marry the aspirations of Abramsky's semantic cube to the powerful proof method of *step-indexed Kripke logical relations*. Specifically, we show how to define a fully abstract logical relation for an ML-like language with recursive types, abstract types, general references and call/cc. Then, we show how, under orthogonal restrictions to the expressive power of our language—namely, the restriction to first-order state and/or the removal of call/cc—we can enhance the proving power of our model in correspondingly orthogonal ways, and we demonstrate this proving power on a range of interesting examples.

Our work builds closely on that of Ahmed, Dreyer, and Rossberg (hereafter, ADR) [3], who gave the first logical relation for modeling a language with both abstract types and higher-order state. We take ADR as a starting point because the concepts underlying that model provide a rich framework in which to explore the impact of various computational effects on relational reasoning. In particular, one of ADR's main contributions was an extension of Pitts and Stark's aforementioned "local invariants" method with the ability to establish properties about local state that *evolve* over time in some controlled fashion. ADR exploited this ability in order to reason about *generative* (or *state-dependent*) ADTs.

The central contribution of our present paper is to observe that the degree of freedom with which local state properties may evolve depends directly on which particular effects are present in the programming language under consideration. In order to expound this observation, we first recast the ADR model in the more familiar terms of *state transition systems* (Section 3). The basic idea is that the "possible worlds" of the ADR model are really state transition systems, wherein each state dictates a potentially different property about the heap, and the transitions between states control how the heap properties are allowed to evolve. Aside from being somewhat simpler than ADR's formulation of possible worlds (which relied on various non-standard anthropomorphic notions like "populations" and "laws"), our formulation highlights the essential notion of a *state transition*, which plays a crucial role in our story.

Next, in Section 4, we explain how to extend the ADR model with support for first-class continuations via the well-studied technique of *biorthogonality* (aka $\top\top$-closure) [18, 28]. The technical details of this extension are fairly straightforward, with the use of biorthogonality turning out to be completely orthogonal (no pun intended) to the other advanced aspects of the ADR model. That said, this is to our knowledge *the first logical-relations model for a language with call/cc and state*. Moreover, a side benefit of biorthogonality is that it renders our model *both sound and complete* w.r.t. observational equivalence (unlike ADR's, which was only sound).[1] Interestingly, nearly all of the example program equivalences proved in the ADR paper continue to hold in the presence of call/cc, and their proofs carry over easily to our present formulation. (There is one odd exception, the "callback with lock" example, for which the ADR proof was very fiddly and *ad hoc*. We investigate this example in great detail, as we describe below.)

The ADR paper also included several interesting examples that their method was *unable* to handle. The unifying theme of these examples is that they rely on the *well-bracketed* nature of computation—*i.e.,* the assumption that control flow follows a stack-like discipline—an assumption that is only valid in the *absence* of call/cc. In Section 5, we consider two simple but novel enhancements to our state-transition-system model—*private transitions* and *inconsistent states*—which are only sound in the absence of call/cc and which correspondingly enable us to prove all of ADR's "well-bracketed examples".

Conversely, in Section 6, we consider the additional reasoning power gained by restricting the language to first-order state. We observe that this restriction enables *backtracking* within a state transition system, and we demonstrate the utility of this feature on several examples.

The above extensions to our basic state-transition-system model are orthogonal to each other, and can be used independently or in combination. One notable example of this is ADR's "callback with lock" equivalence (mentioned above), an equivalence that holds *in the presence of either* higher-order state or call/cc but not both. Using private transitions but no backtracking, we can prove this equivalence in the presence of higher-order state but no call/cc; and using backtracking but no private transitions, we can prove it in the presence of call/cc but only first-order state. Yet another well-known example, due originally to O'Hearn [26], is true only *in the absence of both* higher-order state and call/cc; hence, it should come as no surprise that our novel proof of this example (presented in detail in Section 7.5) involves all three of our model's new features working in tandem.

Most of the paper is presented in an informal, pedagogical style. Indeed, one advantage of our state transition systems is that they lend themselves to clean "visual" proof sketches. In Section 7, we make our proof method formally precise and state some of the key metatheoretic results. Due to space limitations, we only work through the formal proof of one representative example. Detailed proofs of our full abstraction results, as well as all our examples (and more!), appear in the companion technical appendix [8].

In Section 8, we briefly consider how our Kripke logical relations are affected by the addition of *exceptions* to the language. Unlike call/cc, exceptions do not impose restrictions on our state transition systems, but they do require us to account for exceptional behavior in our proofs.

Finally, in Section 9, we compare our methods to related work and suggest some directions for future work.

2. The Language(s) Under Consideration

In its unrestricted form, the language that we consider is a standard polymorphic lambda calculus with existential, pair, and isorecursive types, general references (higher-order state), and first-

[1] It is important to note that the completeness result has nothing to do with the particular features present in the language, and all to do with the use of biorthogonality. In particular, biorthogonality gives us a uniform way of constructing fully abstract models for *all* of the different languages considered in this paper, regardless of whether they contain call/cc, general references, etc. See Section 9 for further discussion of this point.

class continuations (call/cc). We call this language **HOSC**. Its syntax and excerpts of its call-by-value semantics are given in Figure 1. Dots (...) in the syntax cover primitive operations on base types b, such as addition and if-then-else. To ensure unique typing, various constructs have explicit type annotations, which we will typically omit if they are implicit from context. Evaluation contexts K, injected into the term language via $\mathsf{cont}_\tau\ K$, represent first-class continuations. They are a subset of general contexts C ("terms with a hole"), which are not shown here, but are standard. Their typing judgment $\vdash C : (\Sigma; \Delta; \Gamma; \tau) \rightsquigarrow (\Sigma'; \Delta'; \Gamma'; \tau')$ basically says that for any e with $\Sigma; \Delta; \Gamma \vdash e : \tau$ we have $\Sigma'; \Delta'; \Gamma' \vdash C[e] : \tau'$. The continuation typing judgment $\Sigma; \Delta; \Gamma \vdash K \div \tau$ says that K is an evaluation context with a hole of type τ. Finally, contextual (or observational) approximation, written $\Sigma; \Delta; \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau$, means that in any well-typed program context C, if $C[e_1]$ terminates, then so does $C[e_2]$. Contextual (or observational) equivalence is then defined as approximation in both directions.

By restricting **HOSC** in two orthogonal ways, we obtain three fragments of interest:

FOSC The result of restricting to first-order state. Concretely, this means only permitting reference types $\mathsf{ref}\ b$, where b represents base types like int, bool, etc.

HOS The result of removing call/cc, *i.e.,* dropping the type $\mathsf{cont}\ \tau$ and the corresponding three term-level constructs.

FOS The result of making both of the above restrictions.

$$\tau ::= \alpha \mid b \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mu\alpha.\,\tau \mid \mathsf{ref}\,\tau \mid \mathsf{cont}\,\tau$$

$$\begin{aligned} e ::=\ & x \mid l \mid \langle e_1, e_2\rangle \mid e.1 \mid e.2 \mid \lambda x{:}\tau.\,e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau \mid \\ & \mathsf{pack}\ \langle\tau_1, e\rangle\ \mathsf{as}\ \tau_2 \mid \mathsf{unpack}\ e_1\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ e_2 \mid \\ & \mathsf{roll}_\tau\ e \mid \mathsf{unroll}\ e \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid !e \mid e_1 == e_2 \mid \\ & \mathsf{cont}_\tau\ K \mid \mathsf{call/cc}_\tau\,(x.\,e) \mid \mathsf{throw}_\tau\ e_1\ \mathsf{to}\ e_2 \mid \ldots \end{aligned}$$

$$\begin{aligned} K ::=\ & \bullet \mid \langle K, e_2\rangle \mid \langle v_1, K\rangle \mid K.1 \mid K.2 \mid K\ e_2 \mid v_1\ K \mid K\ \tau \mid \\ & \mathsf{pack}\ \langle\tau_1, K\rangle\ \mathsf{as}\ \tau_2 \mid \mathsf{unpack}\ K\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ e_2 \mid \\ & \mathsf{roll}_\tau\ K \mid \mathsf{unroll}\ K \mid \mathsf{ref}\ K \mid K := e_2 \mid v_1 := K \mid !K \mid \\ & K == e_2 \mid v_1 == K \mid \\ & \mathsf{throw}_\tau\ K\ \mathsf{to}\ e_2 \mid \mathsf{throw}_\tau\ v_1\ \mathsf{to}\ K \mid \ldots \end{aligned}$$

$$v ::= x \mid l \mid \langle v_1, v_2\rangle \mid \lambda x{:}\tau.\,e \mid \Lambda\alpha.e \mid \mathsf{pack}\ \langle\tau_1, v\rangle\ \mathsf{as}\ \tau_2 \mid \mathsf{roll}_\tau\ v \mid \mathsf{cont}_\tau\ K \mid \ldots$$

$$\begin{array}{lll} \langle h; K[\mathsf{ref}\ v]\rangle & \hookrightarrow \langle h \uplus \{l \mapsto v\}; K[l]\rangle & (l \notin \mathrm{dom}(h)) \\ \langle h; K[l := v]\rangle & \hookrightarrow \langle h[l \mapsto v]; K[\langle\rangle]\rangle & (l \in \mathrm{dom}(h)) \\ \langle h; K[!l]\rangle & \hookrightarrow \langle h; K[v]\rangle & (h(l) = v) \\ \langle h; K[l_1 == l_2]\rangle & \hookrightarrow \langle h; K[\mathsf{tt}]\rangle & (l_1 = l_2) \\ \langle h; K[l_1 == l_2]\rangle & \hookrightarrow \langle h; K[\mathsf{ff}]\rangle & (l_1 \neq l_2) \\ \langle h; K[\mathsf{call/cc}_\tau\,(x.\,e)]\rangle & \hookrightarrow \langle h; K[e[\mathsf{cont}_\tau\ K/x]]\rangle & \\ \langle h; K[\mathsf{throw}_\tau\ v\ \mathsf{to}\ \mathsf{cont}_{\tau'}\ K']\rangle & \hookrightarrow \langle h; K'[v]\rangle & \end{array}$$

$$\begin{array}{llll} \text{Heap typings} & \Sigma & ::= \cdot \mid \Sigma, l{:}\tau & \text{where } \mathrm{fv}(\tau) = \emptyset \\ \text{Type environments} & \Delta & ::= \cdot \mid \Delta, \alpha & \\ \text{Term environments} & \Gamma & ::= \cdot \mid \Gamma, x{:}\tau & \end{array}$$

$$\frac{\vdash K : (\Sigma; \Delta; \Gamma; \tau) \rightsquigarrow (\Sigma; \Delta; \Gamma; \tau')}{\Sigma; \Delta; \Gamma \vdash K \div \tau} \qquad \frac{\Sigma; \Delta; \Gamma \vdash K \div \tau}{\Sigma; \Delta; \Gamma \vdash \mathsf{cont}_\tau\ K : \mathsf{cont}\,\tau}$$

$$\frac{\Sigma; \Delta; \Gamma, x{:}\mathsf{cont}\,\tau \vdash e : \tau}{\Sigma; \Delta; \Gamma \vdash \mathsf{call/cc}_\tau\,(x.\,e) : \tau} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e' : \tau' \quad \Sigma; \Delta; \Gamma \vdash e : \mathsf{cont}\,\tau'}{\Sigma; \Delta; \Gamma \vdash \mathsf{throw}_\tau\ e'\ \mathsf{to}\ e : \tau}$$

$$\frac{\forall l{:}\tau \in \Sigma.\ \Sigma; \cdot; \cdot \vdash h(l) : \tau}{\vdash h : \Sigma}$$

$$\boxed{\Sigma; \Delta; \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau} \stackrel{\mathrm{def}}{=} \begin{array}{l} \Sigma; \Delta; \Gamma \vdash e_1 : \tau \wedge \Sigma; \Delta; \Gamma \vdash e_2 : \tau \wedge \forall C, \Sigma', \tau', h. \\ \vdash C : (\Sigma; \Delta; \Gamma; \tau) \rightsquigarrow (\Sigma'; \cdot; \cdot; \tau') \wedge \vdash h : \Sigma' \wedge \\ \langle h; C[e_1]\rangle\!\downarrow \implies \langle h; C[e_2]\rangle\!\downarrow \end{array}$$

Figure 1. The Language **HOSC**

3. A Model Based on State Transition Systems

The Ahmed-Dreyer-Rossberg (ADR) model [3], on which our model is based, is a step-indexed Kripke logical relation for the language **HOS**. In this section, we will briefly review what a step-indexed Kripke logical relation is, what is interesting about the ADR model, and how we can recast the essence of the ADR model in terms of *state transition systems*.

Step-Indexed Kripke Logical Relations Logical relations are one of the best-known methods for local reasoning about equivalence (or, more generally, approximation) in higher-order, typed languages. The basic idea is to define the equivalence or approximation relation in question inductively over the type structure of the language, with each type constructor being interpreted by the logical connective to which it corresponds. For instance, two functions are logically related if relatedness of their arguments *implies* relatedness of their results; two existential packages are logically related if there *exists* a relational interpretation of their hidden type representations that is preserved by their operations; and so forth.

In order to reason about equivalence in the presence of state, it becomes necessary to place constraints on the heaps under which programs are evaluated. This is where *Kripke* logical relations come in. Kripke logical relations [28] are logical relations indexed by a *possible world* W, which codifies some set of heap constraints. Roughly speaking, e_1 is related to e_2 under W only if they behave "the same" when run under any heaps h_1 and h_2 that satisfy the constraints of W. When reasoning about programs that maintain some *local* state, possible worlds allow us to impose whatever invariants on the local state we want, so long as we ensure that those invariants are preserved by the code that accesses the state.

To make things concrete, consider the following example:

$$\begin{array}{lcl} \tau & = & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ e_1 & = & \mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in}\ \lambda f.\,(f\ \langle\rangle; !x) \\ e_2 & = & \lambda f.\,(f\ \langle\rangle; 1) \end{array}$$

We would like to show that e_1 and e_2 are observationally equivalent at type τ. The reason, intuitively, is obvious: the reference x is kept private (*i.e.,* it is never leaked to the context), and since it is never modified by the function returned by e_1, it will always point to 1. To prove this using Kripke logical relations, we would set out to prove that e_1 and e_2 are related under an arbitrary initial world W. So suppose we evaluate the two terms under heaps h_1 and h_2 that satisfy W. Since the evaluation of e_1 results in the allocation of some *fresh* memory location for x (*i.e., $x \notin \mathrm{dom}(h_1)$*), we know that the initial world W cannot already contain any constraints governing the contents of x. (If it contained such a constraint, h_1 would have had to satisfy it, and hence x would have to be in $\mathrm{dom}(h_1)$.) So we may extend W with a *new* invariant stating that $x \hookrightarrow 1$ (*i.e.,* x points to 1). It then remains to show that the two λ-abstractions are logically related under this extended world—*i.e.,* under the assumption that $x \hookrightarrow 1$—which is straightforward.

Finally, *step-indexed* logical relations [4, 2] were proposed (originally by Appel and McAllester) as a way to account for se-

mantically problematic features, such as general recursive types, whose relational interpretations are seemingly "cyclic" and thus difficult to define inductively. The idea is simply to stratify the construction of the logical relation by a natural number (or "step index"), representing roughly the number of steps of computation for which the programs in question behave in a related manner.

One of the key contributions of the ADR model was to combine the machinery of step-indexed logical relations with that of Kripke logical relations in order to model higher-order state. While the details of this construction are quite interesting, they are orthogonal to the novel contributions of the model we present in this paper. Indeed, our present model follows ADR's very closely in its use of step-indexing to resolve circularities in the construction, and so we refer the interested reader to the ADR paper for details.

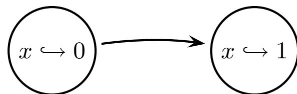
ADR and State Transition Systems The other key contribution of the ADR model was to provide an enhanced notion of possible world, which has the potential to express properties of local state that *evolve* over time. To motivate this feature of ADR, consider a simple variant of the example shown above, in which the first program e_1 is replaced by

$$e_1 \quad = \quad \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda f.\,(x := 1; f\ \langle\rangle; !x)$$

Here, x starts out pointing to 0, but if the function that e_1 evaluates to is ever called, x will be set to 1 and will never change back to 0. In this case, the only invariant one can prove about x is that it points to *either* 0 or 1, but this invariant is insufficient to establish that after the call to the callback f, the contents of x have not changed back to 0. For this reason, Pitts and Stark, whose possible-worlds model only supported heap *invariants*, called this example the "awkward" example (because they could not handle it) [28].

While the awkward example is clearly contrived, it is also a minimal representative of a useful class of programs in which changes to local state occur in some monotonic fashion. As ADR showed, this includes well-known *generative* (or *state-dependent*) ADTs, in which the interpretation of an abstract type grows over time in correspondence with changes to some local state.

ADR's solution was to generalize possible worlds' notion of "heap constraint" to express heap properties that change in a controlled fashion. We can understand their possible worlds as essentially *state transition systems*, where each state determines a particular heap property, and where the transitions determine how the heap property may evolve. For instance, in the case of the awkward example, ADR would represent the heap constraint on x via the following state transition system (STS):



Initially, x points to 0, and then it is set to 1. Since the call to the callback f occurs when we are in the $x \hookrightarrow 1$ state, we know it must return in the same state since there is no transition out of that state. Correspondingly, it is necessary to also show that the $x \hookrightarrow 1$ state is really final—*i.e.,* if the function to which e_1 evaluates is called in that state, it will not change x's contents again—but this is obvious.

In ADR, states are called "populations" and state transition systems are called "laws", but the power of their possible worlds is very similar to that of our STS's (as we have described them thus far), and most of their proofs are straightforwardly presentable in terms of STS's. That said, the two models are not identical. In particular, there is one example we are aware of, the "callback with lock" example, that is provable in ADR's model but not in our basic STS model. As we will see shortly, there are good reasons why this example is not provable in our basic STS model, and in Section 5.1, we will show how to extend our STS's in order to prove this very example in a much simpler, cleaner way than ADR's model does.

4. Biorthogonality, Call/cc, and Full Abstraction

One point on which different formulations of Kripke logical relations differ is the precise formulation of the logical relation for *terms*. The ADR model employs a "direct-style" term relation, which can be described informally as follows: two terms e_1 and e_2 are logically related under world W iff whenever they are evaluated in initial heaps h_1 and h_2 satisfying W, they either both diverge or they both converge to machine configurations $\langle h'_1; v_1\rangle$ and $\langle h'_2; v_2\rangle$ such that $h'_1$and h'_2 satisfy W' and v_1 and v_2 are logically related values under W', where W' is *some* "future world" of W. (By "future world", we mean that W' extends W with new constraints about freshly allocated pieces of the heap, and/or the heap constraints of W may have evolved to different heap constraints in W' according to the STS's in W.) We call this a direct-style term relation because it involves evaluating the terms *directly* to values and then showing relatedness of those values in some future world.

An alternative approach, first employed in the logical relations setting by Pitts and Stark [28] but subsequently adopted by several others (*e.g.,* [13, 7, 5]), is what one might call a "CPS" term relation, although it is more commonly known as a *biorthogonal* (or $\top\top$-closed) term relation. The idea is to define two terms to be related under world W if they co-terminate (both converge or both diverge) when evaluated under heaps that satisfy *W and under continuations K_1 and K_2 related under W*. The latter (continuation relatedness) is then defined to mean that, for any future world W' of W, the continuations K_1 and K_2 co-terminate when applied (under heaps that satisfy W') to *values* that are related under W'. In this way, the logical relation for values is lifted to a logical relation for terms by a kind of CPS transform.

The main arguable advantage of the direct-style term relation is that its definition is perhaps more intuitive, corresponding closely to the proof sketches of the sort that we will present informally in the sections that follow. That said, in any language for which a direct-style relation is sound, it is typically possible to start instead with a biorthogonal relation and then prove a direct-style proof principle—*e.g.,* Pitts and Stark's "principle of local invariants" [28]—as a corollary.

The advantages of the biorthogonal approach are clearer. First, it automagically renders the logical relation *complete* with respect to observational equivalence, largely irrespective of the particular features in the language under consideration. (Actually, it is not so magical: $\top\top$-closure is essentially a kind of closure under observational equivalence.) Second, and perhaps more importantly, the biorthogonal approach scales to handle languages with first-class continuations, such as our **HOSC** and **FOSC**, which the direct-style doesn't. The reason for this is simple: the direct-style approach is only sound if the evaluation of terms is *independent* of the continuation under which they are evaluated. If the terms' behavior is context-dependent, then it does not suffice to consider their co-termination under the empty continuation, which is effectively what the direct-style term relation does. Rather, it becomes necessary to consider co-termination of whole programs (terms together with their continuations), as the biorthogonal relation does.

Thus, in this paper we adopt the biorthogonal approach. This enables us to easily adapt all the proofs from the ADR paper (save for one) to also work for a language with call/cc. (The one exception is the "callback with lock" equivalence, which simply doesn't hold in the presence of call/cc.) It is worth noting that, although the kinds of example programs we focus on in this paper do not involve abstract types, a number of the ADR examples do.

Additionally, we can prove equivalences involving programs that manipulate *both* call/cc and higher-order state. One well-

known challenging example of such an equivalence is the correctness of Friedman and Haynes' encoding of call/cc via "one-shot" continuations (continuations that can only be invoked once) [11, 34]. The basic idea of the encoding is to model an unrestricted continuation using a private (local) ref cell that contains a one-shot continuation. Every time the continuation is invoked, the ref cell is updated with a fresh one-shot continuation. With biorthogonal logical relations, the proof of this example is completely straightforward, employing just a simple invariant on the private ref cell. As far as we know, though, this proof is novel. Full details are given in the technical appendix [8].

5. Reasoning in the Absence of Call/cc

In this section, we examine some reasoning principles that are *enabled* by removing call/cc from our language.

Consider this variant of the "awkward" example (from ADR):

$$\begin{array}{lcl} \tau & = & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int} \\ e_1 & = & \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\ & & \lambda f.\,(x := 0; f\ \langle\rangle; x := 1; f\ \langle\rangle; !x) \\ e_2 & = & \lambda f.\,(f\ \langle\rangle; f\ \langle\rangle; 1) \end{array}$$

What has changed is that now the callback is run twice, and in e_1, the first call to f is preceded by the assignment of x to 0, not 1.

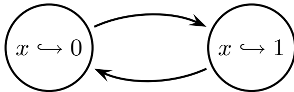
It is easy to see that e_1 and e_2 are not equivalent in **HOSC** (or even **FOSC**). In particular, here is a distinguishing context C:

$$\begin{array}{l} \mathsf{let}\ g = \bullet\ \mathsf{in}\ \mathsf{let}\ b = \mathsf{ref}\ \mathsf{ff}\ \mathsf{in} \\ \mathsf{let}\ f = (\lambda_.\ \mathsf{if}\ !b\ \mathsf{then}\ \mathsf{call/cc}\,(k.\ g\ (\lambda_.\ \mathsf{throw}\ \langle\rangle\ \mathsf{to}\ k)) \\ \qquad\qquad\qquad\quad \mathsf{else}\ b := \mathsf{tt})\ \mathsf{in} \\ g\ f \end{array}$$

Exploiting its ability to capture the continuation K of the second call to f, the context C is able to set x back to 0 and then immediately throw control back to K. It is easy to verify that $C[e_1]$ yields 0, while $C[e_2]$ yields 1.

In the absence of call/cc, however, computations are "well-bracketed". Here, this means that whenever x is set to 0, it will eventually be set to 1—no matter what the callback function does. Consequently, it seems intuitively clear that these programs are equivalent in **HOS** (and **FOS**), but how do we prove it? The STS model we have developed so far will clearly not do the job, precisely because that model is *compatible* with call/cc and this example is not. So the question remains: how can we augment the power of our STS's so that they take advantage of well-bracketing?

To see how to answer this question, let's see what goes wrong if we try to give an STS for our well-bracketed equivalence. First, recall the STS (from Section 3) that we used in order to prove the original awkward example. To see why this STS is insufficient for our present purposes, suppose the function value resulting from evaluating e_1—call it v_1—is applied in the $x \hookrightarrow 1$ state.[2] The first thing that happens is that x is set to 0. However, as there is no transition from the $x \hookrightarrow 1$ state to the $x \hookrightarrow 0$ state, there is no way we can continue the proof. So how about adding that transition?

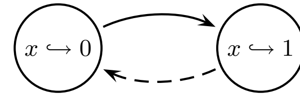


While adding the transition from $x \hookrightarrow 1$ to $x \hookrightarrow 0$ clears the first hurdle, it also erects a new one: according to the STS, it is now possible that, after the second call to f, we end up in the left state—even though this situation (x pointing to 0 after that call) cannot actually arise in reality. And indeed, if x could point to 0 at that point, our proof would be doomed. In summary, while we would like to add this transition, we also want to keep the context from using it. This is where *private transitions* come in.

5.1 Private Transitions

Private transitions are a new class of transitions in our state transition systems, separate from the ordinary transitions that we have seen so far (and which we henceforth call *public transitions*). The basic idea is very simple: when reasoning about the relatedness of terms, we must show that—when viewed *extensionally*—they appear only to be making public transitions, and correspondingly we may assume that the context only makes public transitions as well. Internally, however, *within* a computation, we may make use of both public and private transitions.

Concretely, we can use the following STS to prove our running example (where the dashed arrow denotes a private transition):



First, if v_1 is called in the starting state $x \hookrightarrow 1$, the presence of the private transition allows us to "lawfully" transition from $x \hookrightarrow 1$ to $x \hookrightarrow 0$. Second, we know that, because we are in the $x \hookrightarrow 1$ state before the second call to f and there is no public transition from there to any other state, we must still be in that same state when f returns. Hence we know that x points to 1 at that point, as desired. Lastly, although the body of v_1 makes a private transition internally (when called in starting state $x \hookrightarrow 1$), it appears extensionally to make a public transition, since its final state ($x \hookrightarrow 1$) is obviously publicly accessible from whichever state was the initial one.

Private transitions let us prove not only this example, but also several others from the literature that hold exclusively in the absence of call/cc (including Pitts and Stark's "higher-order profiling" example [28]—see the appendix [8] for details). The intuitive reason why private transitions "don't work" with call/cc is that, in the presence of call/cc, every time we pass control to the context may be the last! Therefore, the requirement that the extensional behavior of a term must appear like a public transition would essentially imply that every internal transition must be public as well.

The "Callback with Lock" Example Here is another equivalence (from ADR) that holds in **HOS** but not in **HOSC**. Interestingly, this example was provable in the original ADR model, but only through some complex step-index hackery. The proof we are about to sketch is much cleaner and easier to understand.

Consider the following two encodings of a counter object with two methods: an *increment* function that also takes a callback argument, which it invokes, and a *poll* function that returns the current counter value.

$$\begin{array}{lcl} C & = & \mathsf{let}\ b = \mathsf{ref}\ \mathsf{tt}\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in} \\ & & \langle \lambda f.\ \mathsf{if}\ !b\ \mathsf{then}\ b := \mathsf{ff}; \bullet; b := \mathsf{tt}\ \mathsf{else}\ \langle\rangle, \\ & & \ \lambda_.\ !x \rangle \\ \tau & = & ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{int}) \\ e_1 & = & C[f\ \langle\rangle; x := !x + 1] \\ e_2 & = & C[\mathsf{let}\ n = !x\ \mathsf{in}\ f\ \langle\rangle; x := n + 1] \end{array}$$

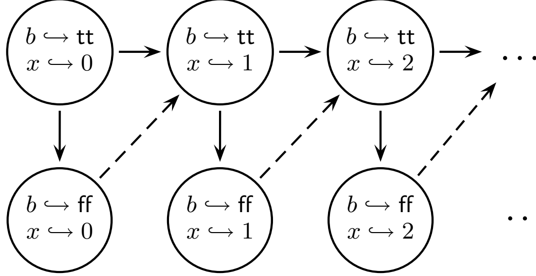
Note that in the second program the counter x is dereferenced *before* the callback is executed, and in the first program it is dereferenced *after*. In both programs, a Boolean lock b guards the increment of the counter, thereby enforcing that running the callback will not result in any change to the counter.

It is not hard to construct a context that exploits the combination of call/cc and higher-order state in order to distinguish e_1 and e_2.

[2] When proving functions logically related, we must consider the possibility that they are invoked in an arbitrary "future" world—*i.e.,* a world where our STS may be in any state that is reachable from its initial state. This ensures *monotonicity* of the logical relation (Theorem 1, Section 7.1).

The basic idea is to pass the increment method a callback that captures its current continuation and stores that in a ref cell so it can be invoked later. The definition of this distinguishing context appears in the appendix [8].

In the absence of call/cc, however, the two programs are equivalent. To prove this, we employ the following infinite STS:



For each number n there are two states: one (the "unlocked" state) saying that b points to tt and x points to n in both programs, and another (the "locked" state) saying that b points to ff and x points to n in both programs. It is thus easy to see that the two poll methods are related (they return the same number). To show the increment methods related, suppose they are executed in a state where x points to some m and b points to tt (the other case where $b \hookrightarrow \mathsf{ff}$ is trivial). Before invoking the callback, b is set to ff and, in the second program, n is bound to m. Accordingly, we move "downwards" in our STS to the locked state and can then call f. Because that state does not have any other public successors, we will still be there if and when f returns—indeed, this is the essence of what it means to be a "locked" state. In the first program, x is then incremented, *i.e.,* set to $m + 1$. In the second program, x is set to $n + 1 = m + 1$. Finally, b is set back to tt and we thus move to the matching private successor ($b \hookrightarrow \mathsf{tt}$, $x \hookrightarrow m + 1$) in the STS. Since this is a public successor of the initial state ($b \hookrightarrow \mathsf{tt}$, $x \hookrightarrow m$), our extensional transition appears public and we are done.

5.2 Inconsistent States

While private transitions are clearly a useful extension to our STS model, there is one kind of "well-bracketed example" we are aware of that private transitions alone are insufficient to account for. We are referring to the "deferred divergence" example, presented by ADR as an example they could not handle. The original version of this equivalence, due to O'Hearn [26], was presented in the setting of Idealized Algol, and it does not hold in the presence of higher-order state. (We will consider a variant of O'Hearn's example later on, in Section 6.) Here, we consider a version of the equivalence that *does* hold in **HOS**, based on the one in Bohr's thesis [7]:

$$\begin{array}{rcl}
\tau & = & ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \\
e_1 & = & \mathsf{let}\ x = \mathsf{ref\ ff\ in\ let}\ y = \mathsf{ref\ ff\ in} \\
& & \lambda f.\, f\ (\lambda_.\ \mathsf{if}\ !x\ \mathsf{then}\ \bot\ \mathsf{else}\ y := \mathsf{tt}); \\
& & \quad \mathsf{if}\ !y\ \mathsf{then}\ \bot\ \mathsf{else}\ x := \mathsf{tt} \\
e_2 & = & \lambda f.\, f\ (\lambda_.\ \bot)
\end{array}$$

Intuitively, the explanation why e_1 and e_2 are equivalent goes as follows. The functions returned by both programs take a higher-order callback f as an argument and apply it to a thunk. In the case of e_2, if that thunk argument ($\lambda_.\ \bot$, where $\bot$ is a divergent term) is ever applied, either during the call to f or at some point in the future (*e.g.,* if the thunk were stored by f in a ref cell and then called later), then the program will clearly diverge. Now, e_1 implements the same divergence behavior, but in a rather sneaky way. It maintains two private flags x and y, initially set to ff. If the thunk that it passes to f is applied *during* the call to f, then the thunk's body will not immediately diverge (as in the case of e_2), but rather merely set y to tt. Then, if and when f returns, e_1 will check if y points to tt and, if so, diverge. If the thunk was not applied during the call to f, then e_1 will set x to tt, thus ensuring that any future attempt to apply the thunk will diverge as well.

As in the previous examples, note that this equivalence does not hold in the presence of call/cc. Here is a distinguishing context:

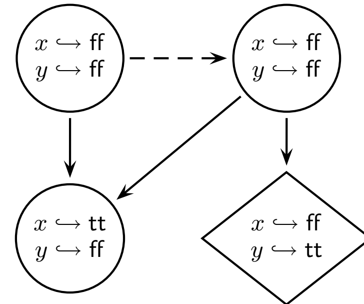
$$\mathsf{call/cc}\,(k.\ \bullet\ (\lambda g.\,\mathsf{throw}\ g\ \langle\rangle\ \mathsf{to}\ k))$$

To prove the equivalence in **HOS**, we can split the proof into two directions of approximation. Proving that e_2 approximates e_1 is actually very easy because (1) it is trivial to show that $\lambda_.\ \bot$ approximates the thunk that e_1 passes to f, and (2) if a program $C[e_2]$ terminates (which is the assumption of observational approximation), then $C[e_1]$ must in fact maintain the invariant that $y \hookrightarrow \mathsf{ff}$, and using that invariant the proof is totally straightforward.

In contrast, the other direction of approximation seems at first glance impossible to prove using logical relations. The issue is that we have to show that the thunks passed to the callback f are related, *i.e.,* that $\lambda_.\ \mathsf{if}\ !x\ \mathsf{then}\ \bot\ \mathsf{else}\ y := \mathsf{tt}$ approximates $\lambda_.\ \bot$, which obviously is *false* since, when applied (as they may be) in a state where x points to ff, the first converges while the second diverges.

To solve this conundrum, we do the blindingly obvious thing, which is to introduce *falsehood* into our model! Specifically, we extend our STS's with *inconsistent states*, in which we can prove false things, such as that a terminating computation approximates a divergent one. How, one may ask, can this possibly work? The idea is as follows: when we enter an inconsistent state, we effectively shift the proof burden from the logical relation for terms to the logical relation for *continuations*. That is, while it becomes very easy to prove that two terms are related in an inconsistent state, it becomes very *hard* to prove that two continuations K_1 and K_2 are related in such a state—in most cases, we will be forced to prove that K_1 diverges. Thus, while inconsistent states do allow a limited kind of falsehood inside an approximation proof, we can only enter into them if we *know* that the continuation of the term on the left-hand side of the approximation will diverge anyway.

Concretely, to show that e_1 approximates e_2, we construct the following STS, where the diamond indicates an inconsistent state:



For the moment, ignore the top-left state (we explain it below). In the proof, we wish to show that the thunks passed to the callback f are logically related in the top-right state, which requires showing that they are related in any state accessible from it. Fortunately, this is easy. If the thunks are called in the bottom-left state, then they both diverge. If they are called in the top-right or bottom-right state, then the else-branch is executed (in the first program) and we move to (or stay in) the bottom-right state—since this state is inconsistent, the proof is trivially done.

Dually, we must show that the continuations of the callback applications are also related in any state (publicly) accessible from the top-right one. If the continuations are invoked in the top-right or the bottom-left state, they will set x to tt, thereby transitioning to the bottom-left. If, on the other hand, they are invoked in the inconsistent bottom-right state, then we are required to show that the first one diverges, which fortunately it will since y points to tt.

Now about the top-left state, whose heap constraint is identical to the one in the top-right state: the reason for including this state has to do with soundness of the logical relation. In order to ensure soundness, we require that when an STS is installed in the possible world, it may not contain any inconsistent states that are *publicly* accessible from its starting state. We say in this case that the starting state is *safe*. (Without this safety restriction, it would be easy to show, for instance, that tt approximates ff in any world W by simply adding an STS to W with a single inconsistent state.)

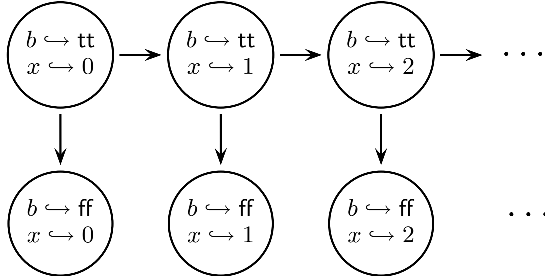
To circumvent this restriction, we use the top-left state as our starting state and connect it to the top-right state by a private transition. (In the proof, the first step before invoking the callbacks is to transition into the top-right state.) This is fine so long as the extensional behavior of the functions we are relating makes a public transition, and here it does—if they are invoked in the top-left state, then either they diverge or they return control in the bottom-left state, which is publicly accessible from the top-left.

6. Reasoning With First-Order State

In this section, we consider an orthogonal restriction to the one examined in the previous section. Instead of removing call/cc from the language, what happens if we restrict state to be first-order? What new reasoning principles are enabled by this restriction?

6.1 Backtracking

Recall the "callback with lock" example from Section 5.1, which we proved equivalent in **HOS**. As it turns out, that equivalence also holds in **FOSC**. Of course, we won't be able to prove that using the **HOSC** model since the equivalence doesn't hold in **HOSC**. But let us see what exactly goes wrong if we try. First of all, recall the use of private transitions in our earlier proof. Due to call/cc, we cannot use any private transitions this time. Clearly, making them public is not an option, so what if we just drop them entirely?



In the resulting STS, we still know that running the callback in a locked state ($b \hookrightarrow$ ff, $x \hookrightarrow m$) will leave us in the very same state if and when it returns. However, without any outgoing (private) transition from that state, it seems that we are subsequently stuck.

Fortunately, we are not. The insight now is that the absence of higher-order state allows us to do *backtracking* within our STS. Concretely, we can backtrack from the locked state to the unlocked state we were in before ($b \hookrightarrow$ tt, $x \hookrightarrow m$), and then transition (publicly) to its successor ($b \hookrightarrow$ tt, $x \hookrightarrow m + 1$). Intuitively, this kind of backtracking would not be sound in the presence of higher-order state because, in that setting, the callback might have stored some higher-order data during its execution (such as functions or continuations) that are only logically related in the locked state and its successors.[3] Since ($b \hookrightarrow$ tt, $x \hookrightarrow m + 1$) is not a successor of the previous locked state, the final heaps would then fail to satisfy the final world in which the increment functions return. Here in the first-order setting, though, there is no way for the callback to store such higher-order data, so backtracking is not a problem. A precise technical explanation of how the model is changed to allow backtracking, and why this is sound, will be given in Section 7.3.

6.2 Putting It Together

The example we just looked at might suggest that backtracking is mainly useful as a replacement for private transitions in the presence of call/cc. But in fact, they are complementary techniques. In particular, for equivalences that hold only in **FOS** but not in **HOS** or **FOSC**, we can profitably employ backtracking, private transitions, and inconsistent states, all working together.

Consider this simpler version of the "deferred divergence" example, based closely on an example of O'Hearn [26]:

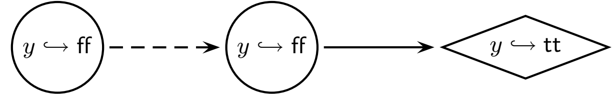
$$\begin{aligned}
\tau &= ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \\
e_1 &= \mathsf{let}\ y = \mathsf{ref}\ \mathsf{ff}\ \mathsf{in} \\
&\quad\ \lambda f.\, f\ (\lambda_.\, y := \mathsf{tt}); \\
&\quad\quad \mathsf{if}\ !y\ \mathsf{then}\ \bot\ \mathsf{else}\ \langle\rangle \\
e_2 &= \lambda f.\, f\ (\lambda_.\, \bot)
\end{aligned}$$

These programs are not only distinguishable in the setting of **FOSC** (by the same distinguishing context as given in Section 5.2), but also in **HOS**, as the following context demonstrates:

$$C \quad = \quad \mathsf{let}\ r = \mathsf{ref}\ (\lambda_.\, \langle\rangle)\ \mathsf{in}\ \bullet\ (\lambda g.\, r := g); !r\ \langle\rangle$$

It is easy to verify that $C[e_1]$ terminates, while $C[e_2]$ diverges.

The two programs are, however, equivalent in **FOS**, which we can prove using the following STS:



The proof is largely similar to (if a bit simpler than) the one sketched for the higher-order version of this example in Section 5.2. We start in the left state and transition immediately along the private transition to the middle state. With the help of the inconsistent right state, it is easy to show that the thunk arguments passed to the callback are related in the middle state. Hence, when the callback returns, we are either in the right state or the middle state. In the former case, we must show that the continuation in the l.h.s. program diverges; in the latter, we *backtrack* to the initial, left state, which is of course publicly accessible from itself. (We will present this proof in more detail below, in Section 7.5.)

Why, one might ask, is it not possible to avoid the use of backtracking here by adding a private transition back from the middle state to the left state? (Of course, it *must* not be possible, or else the equivalence would hold true in **HOS**, which as we have seen it does not.) The answer is that, if we were to add such a transition, then we would not be able to prove that the thunk arguments to the callback f were logically related in the middle state. Specifically, in order to show the latter, we must show that the thunks are related in any state accessible (by any kind of transition) from the middle state. So if there were any transition from the middle to the left state, we would have to show that the thunks were related starting in the left state as well—but they are not, because there is no public transition from the initial left state to the inconsistent right state, and adding one would be unsound.

7. Technical Development

We now present the models for our various languages formally. It is easiest to start with the model for **HOS**, and then show how small changes to that yield the models for **HOSC**, **FOS**, and **FOSC**.

[3] Indeed, the context that distinguishes between the two programs in **HOSC** employs precisely such a callback, namely one that stores its current continuation in a ref cell.

$$
\begin{array}{rcl}
\mathrm{HeapAtom}_n & \stackrel{\text{def}}{=} & \{(W, h_1, h_2) \mid W \in \mathrm{World}_n\} \\
\mathrm{HeapRel}_n & \stackrel{\text{def}}{=} & \{\psi \subseteq \mathrm{HeapAtom}_n \mid \forall (W, h_1, h_2) \in \psi.\ \forall W' \sqsupseteq W.\ (W', h_1, h_2) \in \psi\} \\
\mathrm{Island}_n & \stackrel{\text{def}}{=} & \{\iota = (s, \delta, \varphi, \lightning, H) \mid s \in \mathrm{State} \wedge \delta \subseteq \mathrm{State}^2 \wedge \varphi \subseteq \delta \wedge \delta, \varphi \text{ reflexive} \wedge \delta, \varphi \text{ transitive} \wedge \\
 & & \qquad \lightning \subseteq \mathrm{State} \wedge H \in \mathrm{State} \rightarrow \mathrm{HeapRel}_n\} \\
\mathrm{World}_n & \stackrel{\text{def}}{=} & \{W = (k, \Sigma_1, \Sigma_2, \omega) \mid k < n \wedge \exists m.\ \omega \in \mathrm{Island}_k^m\} \\
\mathrm{ContAtom}_n[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \{(W, K_1, K_2) \mid W \in \mathrm{World}_n \wedge W.\Sigma_1; \cdot; \cdot \vdash K_1 \div \tau_1 \wedge W.\Sigma_2; \cdot; \cdot \vdash K_2 \div \tau_2\} \\
\mathrm{TermAtom}_n[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \{(W, e_1, e_2) \mid W \in \mathrm{World}_n \wedge W.\Sigma_1; \cdot; \cdot \vdash e_1 : \tau_1 \wedge W.\Sigma_2; \cdot; \cdot \vdash e_2 : \tau_2\} \\
\mathrm{ValRel}[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \{r \subseteq \mathrm{TermAtom}^{\mathsf{val}}[\tau_1, \tau_2] \mid \forall (W, v_1, v_2) \in r.\ \forall W' \sqsupseteq W.\ (W', v_1, v_2) \in r\} \\
\mathrm{SomeValRel} & \stackrel{\text{def}}{=} & \{R = (\tau_1, \tau_2, r) \mid r \in \mathrm{ValRel}[\tau_1, \tau_2]\}
\end{array}
$$

$$
\begin{array}{rclcrcl}
\lfloor (\iota_1, \ldots, \iota_m) \rfloor_k & \stackrel{\text{def}}{=} & (\lfloor \iota_1 \rfloor_k, \ldots, \lfloor \iota_m \rfloor_k) & \qquad & \lfloor H \rfloor_k & \stackrel{\text{def}}{=} & \lambda s. \lfloor H(s) \rfloor_k \\
\lfloor (s, \delta, \varphi, \lightning, H) \rfloor_k & \stackrel{\text{def}}{=} & (s, \delta, \varphi, \lightning, \lfloor H \rfloor_k) & & \lfloor \psi \rfloor_k & \stackrel{\text{def}}{=} & \{(W, h_1, h_2) \in r \mid W.k < k\}
\end{array}
$$

$$
\triangleright(k+1, \Sigma_1, \Sigma_2, \omega) \stackrel{\text{def}}{=} (k, \Sigma_1, \Sigma_2, \lfloor \omega \rfloor_k) \qquad \triangleright r \stackrel{\text{def}}{=} \{(W, e_1, e_2) \mid W.k > 0 \implies (\triangleright W, e_1, e_2) \in r\}
$$

$$
\begin{array}{rcl}
(k', \Sigma_1', \Sigma_2', \omega') \sqsupseteq (k, \Sigma_1, \Sigma_2, \omega) & \stackrel{\text{def}}{=} & k' \leq k \wedge \Sigma_1' \supseteq \Sigma_1 \wedge \Sigma_2' \supseteq \Sigma_2 \wedge \omega' \sqsupseteq \lfloor \omega \rfloor_{k'} \\
(\iota_1', \ldots, \iota_{m'}') \sqsupseteq (\iota_1, \ldots, \iota_m) & \stackrel{\text{def}}{=} & m' \geq m \wedge \forall j \in \{1, \ldots, m\}.\ \iota_j' \sqsupseteq \iota_j \\
(s', \delta', \varphi', \lightning', H') \sqsupseteq (s, \delta, \varphi, \lightning, H) & \stackrel{\text{def}}{=} & (\delta', \varphi', \lightning', H') = (\delta, \varphi, \lightning, H) \wedge (s, s') \in \delta
\end{array}
$$

$$
\begin{array}{rcl}
(k', \Sigma_1', \Sigma_2', \omega') \sqsupseteq^{\mathsf{pub}} (k, \Sigma_1, \Sigma_2, \omega) & \stackrel{\text{def}}{=} & k' \leq k \wedge \Sigma_1' \supseteq \Sigma_1 \wedge \Sigma_2' \supseteq \Sigma_2 \wedge \omega' \sqsupseteq^{\mathsf{pub}} \lfloor \omega \rfloor_{k'} \\
(\iota_1', \ldots, \iota_{m'}') \sqsupseteq^{\mathsf{pub}} (\iota_1, \ldots, \iota_m) & \stackrel{\text{def}}{=} & m' \geq m \wedge \forall j \in \{1, \ldots, m\}.\ \iota_j' \sqsupseteq^{\mathsf{pub}} \iota_j \wedge \\
 & & \forall j \in \{m+1, \ldots, m'\}.\ \mathrm{safe}(\iota_j') \\
(s', \delta', \varphi', \lightning', H') \sqsupseteq^{\mathsf{pub}} (s, \delta, \varphi, \lightning, H) & \stackrel{\text{def}}{=} & (\delta', \varphi', \lightning', H') = (\delta, \varphi, \lightning, H) \wedge (s, s') \in \varphi
\end{array}
$$

$$
\mathrm{safe}(W) \stackrel{\text{def}}{=} \forall \iota \in W.\omega.\ \mathrm{safe}(\iota) \qquad \mathrm{safe}(\iota) \stackrel{\text{def}}{=} \forall s'.\ (\iota.s, s') \in \iota.\varphi \implies s' \notin \iota.\lightning \qquad \mathrm{consistent}(W) \stackrel{\text{def}}{=} \nexists \iota \in W.\omega.\ \iota.s \in \iota.\lightning
$$

$$
\psi \otimes \psi' \stackrel{\text{def}}{=} \{(W, h_1 \uplus h_1', h_2 \uplus h_2') \mid (W, h_1, h_2) \in \psi \wedge (W, h_1', h_2') \in \psi'\}
$$

$$
(h_1, h_2) : W \stackrel{\text{def}}{=} \vdash h_1 : W.\Sigma_1 \wedge \vdash h_2 : W.\Sigma_2 \wedge (W.k > 0 \implies (\triangleright W, h_1, h_2) \in \textstyle\bigotimes_{i=1}^{|W.\omega|} W.\omega(i).H(W.\omega(i).s))
$$

Figure 2. Worlds and Auxiliary Definitions

7.1 HOS

As described in Section 3, we employ a step-indexed Kripke logical relation, which is a kind of possible-worlds model.

Worlds Figure 2 displays the construction of worlds, along with various related operations and relations.[4] Worlds W consist of a step index k, heap typings Σ_1 and Σ_2 (for the first and second programs, respectively), and an array of islands $\omega = \iota_1, \ldots, \iota_n$. Islands in turn are (possibly infinite) state transition systems governing disjoint pieces of the heap. Each consists of a current state s, a transition relation δ, a public transition relation φ, a set of inconsistent states $\lightning$, and last but not least, a mapping H from states to heap constraints (in the form of world-indexed heap relations—more on that below). The public transition relation φ must be a subset of the "full" transition relation δ (note: the private transitions are obtained by subtracting φ from δ), and we require both δ and φ to be reflexive and transitive.

What exactly "states" s are—*i.e.,* how we define the state space State—does not really matter. That is, State is essentially a parameter of the model, except that it needs to be at least large enough to encode bijections on memory locations (see our relational interpretation of ref types below). For our purposes, we find it convenient to assume that State contains all terms and all sets of terms. Also, note that while an island's H map is defined on *all* states in State, we typically only care about how it is defined on a particular set of "states of interest"—whether there is other junk in the State space is irrelevant.

Our use of step-indexing to stratify the construction of worlds and to define the logical relation by a primary induction on natural numbers follows the development in ADR quite closely. For space reasons, we therefore omit explanation of the approximation operation $\lfloor \cdot \rfloor_k$, the "later" operator $\triangleright$, and other step-related technicalities and refer the interested reader to the literature [3, 9]. One point about notation, though: we sometimes write World to mean $\bigcup_n \mathrm{World}_n$, and similarly for the other semantic classes.

Based on the two transition relations (full and public), we define two notions of future worlds (aka world extension). First, we say that W' *extends* W, written $W' \sqsupseteq W$, iff it contains the same islands as W (and possibly more), and for each island in W, the new state s' of that island in W'—which is the only aspect of the island that is permitted to change in future worlds—is accessible from the old state s in W, according to the island's full transition relation δ. *Public extension*, written $W' \sqsupseteq^{\mathsf{pub}} W$, is defined analogously, except using the public transition relation φ instead of δ, and with the additional requirement that the new islands (those in W' but not in W) must be *safe*. An island is safe iff there is no public transition from its current state to any inconsistent state.

The reason why our (and ADR's) heap relations are world-indexed is that, when expressing heap constraints, we want to be able to say, for instance, that a value in the first heap must be logically related to a value in the second heap. In that case, we need to have some way of talking about the "current" world under which that logical relation should be considered, and by world-indexing the heap relations we enable the current world to be passed in as a

[4] Here and in the following development we use the dot-notation to project components out of a structure. As an example, we write $W.\Sigma_1$ to extract the first heap typing out of a world W.

$$
\begin{array}{rcl}
\mathcal{V}[\![\alpha]\!]\rho & \stackrel{\text{def}}{=} & \rho(\alpha).r \\
\mathcal{V}[\![b]\!]\rho & \stackrel{\text{def}}{=} & \{(W,v,v) \in \mathrm{TermAtom}[b,b]\} \\
\mathcal{V}[\![\tau \times \tau']\!]\rho & \stackrel{\text{def}}{=} & \{(W, \langle v_1, v_1'\rangle, \langle v_2, v_2'\rangle) \in \mathrm{TermAtom}[\rho_1(\tau \times \tau'), \rho_2(\tau \times \tau')] \mid (W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho \wedge (W, v_1', v_2') \in \mathcal{V}[\![\tau']\!]\rho\} \\
\mathcal{V}[\![\tau' \to \tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, \lambda x{:}\tau_1.\, e_1, \lambda x{:}\tau_2.\, e_2) \in \mathrm{TermAtom}[\rho_1(\tau' \to \tau), \rho_2(\tau' \to \tau)] \mid \\
& & \quad \forall W', v_1, v_2.\, W' \sqsupseteq W \wedge (W', v_1, v_2) \in \mathcal{V}[\![\tau']\!]\rho \implies (W', e_1[v_1/x], e_2[v_2/x]) \in \mathcal{E}[\![\tau]\!]\rho\} \\
\mathcal{V}[\![\forall\alpha.\, \tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, \Lambda\alpha.e_1, \Lambda\alpha.e_2) \in \mathrm{TermAtom}[\rho_1(\forall\alpha.\, \tau), \rho_2(\forall\alpha.\, \tau)] \mid \\
& & \quad \forall W' \sqsupseteq W.\, \forall(\tau_1, \tau_2, r) \in \mathrm{SomeValRel}.\, (W', e_1[\tau_1/\alpha], e_2[\tau_2/\alpha]) \in \mathcal{E}[\![\tau]\!]\rho, \alpha{\mapsto}(\tau_1, \tau_2, r)\} \\
\mathcal{V}[\![\exists\alpha.\, \tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, \mathsf{pack}\ \langle \tau_1, v_1\rangle\ \mathsf{as}\ \tau_1', \mathsf{pack}\ \langle \tau_2, v_2\rangle\ \mathsf{as}\ \tau_2') \in \mathrm{TermAtom}[\rho_1(\exists\alpha.\, \tau), \rho_2(\exists\alpha.\, \tau)] \mid \\
& & \quad \exists r.\, (\tau_1, \tau_2, r) \in \mathrm{SomeValRel} \wedge (W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho, \alpha{\mapsto}(\tau_1, \tau_2, r)\} \\
\mathcal{V}[\![\mu\alpha.\, \tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, \mathsf{roll}_{\tau_1}\ v_1, \mathsf{roll}_{\tau_2}\ v_2) \in \mathrm{TermAtom}[\rho_1(\mu\alpha.\, \tau), \rho_2(\mu\alpha.\, \tau)] \mid (W, v_1, v_2) \in \triangleright\mathcal{V}[\![\tau[\mu\alpha.\, \tau/\alpha]]\!]\rho\} \\
\mathcal{V}[\![\mathsf{ref}\ \tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, l_1, l_2) \in \mathrm{TermAtom}[\rho_1(\mathsf{ref}\ \tau), \rho_2(\mathsf{ref}\ \tau)] \mid \exists i.\, \forall W' \sqsupseteq W.\, (l_1, l_2) \in \mathrm{bij}(W'.\omega(i).s) \wedge \\
& & \quad \exists\psi.\, W'.\omega(i).H(W'.\omega(i).s) = \psi \otimes \{(\widetilde{W}, \{l_1{\mapsto}v_1\}, \{l_2{\mapsto}v_2\}) \in \mathrm{HeapAtom} \mid (\widetilde{W}, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho\}\} \\
\\
\mathcal{O} & \stackrel{\text{def}}{=} & \{(W, e_1, e_2) \mid \forall h_1, h_2.\, (h_1, h_2) : W \wedge \langle h_1; e_1\rangle \downarrow^{<W.k} \implies \mathrm{consistent}(W) \wedge \langle h_2; e_2\rangle \downarrow\} \\
\\
\mathcal{K}[\![\tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, K_1, K_2) \in \mathrm{ContAtom}[\rho_1(\tau), \rho_2(\tau)] \mid \\
& & \quad \forall W', v_1, v_2.\, W' \sqsupseteq^{\mathsf{pub}} W \wedge (W', v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho \implies (W', K_1[v_1], K_2[v_2]) \in \mathcal{O}\} \\
\\
\mathcal{E}[\![\tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, e_1, e_2) \in \mathrm{TermAtom}[\rho_1(\tau), \rho_2(\tau)] \mid \forall K_1, K_2.\, (W, K_1, K_2) \in \mathcal{K}[\![\tau]\!]\rho \implies (W, K_1[e_1], K_2[e_2]) \in \mathcal{O}\}
\end{array}
$$

$$
\begin{array}{rclrcl}
\mathcal{G}[\![\cdot]\!]\rho & \stackrel{\text{def}}{=} & \{(W, \emptyset) \mid W \in \mathrm{World}\} & \mathcal{G}[\![\Gamma, x{:}\tau]\!]\rho & \stackrel{\text{def}}{=} & \{(W, (\gamma, x{\mapsto}(v_1, v_2))) \mid (W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho \wedge (W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho\} \\
\mathcal{D}[\![\cdot]\!] & \stackrel{\text{def}}{=} & \{\emptyset\} & \mathcal{D}[\![\Delta, \alpha]\!] & \stackrel{\text{def}}{=} & \{\rho, \alpha{\mapsto}R \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge R \in \mathrm{SomeValRel}\} \\
\mathcal{S}[\![\cdot]\!] & \stackrel{\text{def}}{=} & \mathrm{World} & \mathcal{S}[\![\Sigma, l{:}\tau]\!] & \stackrel{\text{def}}{=} & \mathcal{S}[\![\Sigma]\!] \cap \{W \in \mathrm{World} \mid (W, l, l) \in \mathcal{V}[\![\mathsf{ref}\ \tau]\!]\emptyset\}
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma; \Delta; \Gamma \vdash e_1 \precsim_{\log} e_2 : \tau & \stackrel{\text{def}}{=} & \Sigma; \Delta; \Gamma \vdash e_1 : \tau \wedge \Sigma; \Delta; \Gamma \vdash e_2 : \tau \wedge \\
& & \forall W, \rho, \gamma.\, W \in \mathcal{S}[\![\Sigma]\!] \wedge \rho \in \mathcal{D}[\![\Delta]\!] \wedge (W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho \implies (W, \rho_1\gamma_1 e_1, \rho_2\gamma_2 e_2) \in \mathcal{E}[\![\tau]\!]\rho
\end{array}
$$

Figure 3. A Step-Indexed Biorthogonal Kripke Logical Relation for **HOS**

parameter. These world-indexed heap relations are quite restricted, however. Specifically, they must be monotone with respect to world extension, meaning that heaps related in one world will continue to be related in any future world. This ensures that adding a new island to the world, or making (any kind of) transition within an existing island, does not violate the heap constraints of other islands.

The last two definitions also concern heap relations. Two heaps h_1 and h_2 *satisfy a world* W, written $(h_1, h_2) : W$, iff they can be split into disjoint subheaps such that for each island in W there is a subheap of h_1 and a corresponding subheap of h_2 that are related by that island's current heap relation (the relation associated with the island's current state). A heap relation ψ is the *tensor* of ψ' and ψ'', written $\psi' \otimes \psi''$, if it contains all (W, h_1, h_2) that can be split into disjoint parts $(W, h_1', h_2') \in \psi'$ and $(W, h_1'', h_2'') \in \psi''$.

Logical Relation Our logical relation for **HOS** is defined in Figure 3. The value relation $\mathcal{V}[\![\tau]\!]\rho$ (where $\mathrm{fv}(\tau) \subseteq \mathrm{dom}(\rho)$) is fairly standard. The only real difference from the ADR model is in $\mathcal{V}[\![\mathsf{ref}\ \tau]\!]\rho$, our interpretation of reference types. Basically, we say that two references l_1 and l_2 are logically related at type $\mathsf{ref}\ \tau$ in world W if there exists an island ι in W, such that (1) ι's heap constraint (in any reachable state) requires of l_1 and l_2 precisely that their contents are related at type τ, and (2) the reachable states in ι encode a bijection between locations that includes the pair (l_1, l_2). The latter condition, which employs an auxiliary "bij" function (defined in the appendix [8]), is needed in order to model the presence of reference equality testing $l_1 == l_2$ in the language. Our formulation of $\mathcal{V}[\![\mathsf{ref}\ \tau]\!]\rho$ is slightly different from ADR's and a bit more flexible—*e.g.,* ours can be used to prove Bohr's "local state release" example [7] (see the appendix), whereas ADR's can't—but this added flexibility does not affect any of our "headlining" examples from Sections 3–6. We will report on the advantages of our present formulation in a future, extended version of this paper.

In logical relations proofs, we frequently assume that we are given some related values (*e.g.,* as inputs to functions), and we want them to be still related after we have added an island to the world or made a transition. It is therefore crucial that, like heap relations, value relations are monotone w.r.t. world extension. Since we enforce this property for relational interpretations of abstract types (see the definition of ValRel in Figure 2), it is easy to show that the value relation indeed has this property:

Theorem 1 (Monotonicity of the Value Relation)**.** If $W' \sqsupseteq W$ and $(W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho$, then $(W', v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho$.

As explained in Section 4, the value relation is lifted to a term relation via biorthogonality. Concretely, we define the continuation relation $\mathcal{K}[\![\tau]\!]\rho$ based on $\mathcal{V}[\![\tau]\!]\rho$, and then the term relation $\mathcal{E}[\![\tau]\!]\rho$ based on $\mathcal{K}[\![\tau]\!]\rho$:

- Two continuations are related iff they yield related observations when applied to related values.
- Two terms are related iff they yield related observations when evaluated under related continuations.

Yielding related observations here means (see the definition of $\mathcal{O}$) that, whenever two heaps satisfy the world W in question and the first program terminates in the first heap (within $W.k$ steps), then the second program terminates in the second heap and the world is *consistent* (*i.e.,* no island is in an inconsistent state). This corresponds to the intuition given in Section 5.2 that an inconsistent world is one in which the first program diverges.

Notice that the continuation relation quantifies only over *public* future worlds. This captures the essential idea (explained in Section 5.1) that the context can only make public transitions. In order to see this, it is important to understand how a typical proof in a biorthogonal logical relation goes. Roughly, showing the related-

ness of two programs that involve a call to an unknown function (*e.g.,* a callback) eventually reduces to showing that the continuations of the function call are related; thanks to the definition of $\mathcal{K}[\![\tau]\!]\rho$, we will only need to consider the possibility that those continuations are invoked in a *public* future world of the world we were in prior to the function call—in other words, we can assume that the function call made a public transition. We will see how this works in detail in the example proof in Section 7.5.

Finally, the logical relation is lifted to open terms in the usual way, quantifying over related closing substitutions δ and γ matching Δ and Γ, respectively, as well as an initial world in which every location bound in Σ is related to itself. We write δ_1 (resp. γ_1) and δ_2 (resp. γ_2) here as shorthand for the first and second type (resp. value) substitutions contained in δ (resp. γ).

Soundness and Completeness The proof that our logical relation is sound w.r.t. contextual approximation follows closely that of ADR [3]. It involves proving the usual "compatibility" lemmas and the construction of a canonical *safe* world for a given heap typing. Details can be found in the technical appendix [8].

Theorem 2 (Fundamental Property)**.** If $\Sigma; \Delta; \Gamma \vdash e : \tau$, then $\Sigma; \Delta; \Gamma \vdash e \precsim_{\text{log}} e : \tau$.

Theorem 3 (Soundness)**.** $\precsim_{\text{log}} \subseteq \precsim_{\text{ctx}}$

Following Pitts and Stark [28], we show completeness of our logical relation w.r.t. contextual approximation with the help of Mason and Talcott's *ciu*-approximation [23] as an intermediate relation.

Theorem 4 (Completeness)**.** $\precsim_{\text{ctx}} \subseteq \precsim_{\text{ciu}} \subseteq \precsim_{\text{log}}$

Proving the inclusion of $\precsim_{\text{ctx}}$ in $\precsim_{\text{ciu}}$ is fairly easy. The inclusion of $\precsim_{\text{ciu}}$ in $\precsim_{\text{log}}$ follows as an almost immediate consequence of the Fundamental Property, together with the logical relation's biorthogonal definition. Again, full details can be found in the appendix [8].

7.2 HOSC

The model for **HOSC** can be obtained from the one for **HOS** by making two changes. First of all, in **HOSC**, we have to account for the presence of first-class continuation values $\mathsf{cont}_\tau\ K$. Fortunately, we already have a continuation relation $\mathcal{K}[\![\tau]\!]\rho$, so it is easy to define the value relation at type $\mathsf{cont}\,\tau$ in terms of it:

$$\mathcal{V}[\![\mathsf{cont}\,\tau]\!]\rho \stackrel{\text{def}}{=} \{(W, \mathsf{cont}\,K_1, \mathsf{cont}\,K_2) \mid (W, K_1, K_2) \in \mathcal{K}[\![\tau]\!]\rho\}$$

Now, recall that we need our value relation to be monotone w.r.t. $\sqsupseteq$. Given the extension we have just made to the value relation for $\mathsf{cont}\,\tau$, that means we need our continuation relation to be monotone w.r.t. $\sqsupseteq$ as well. However, as explained above, the continuation relation is only monotone w.r.t. $\sqsupseteq^{\mathsf{pub}}$ (in order to ensure that the context can only make public transitions). Of course, what this means is that in the presence of call/cc, the private and public transition relations must be collapsed into one, and consequently we must disallow inconsistent states, too. This corresponds to the intuition we gave in Section 5.1, namely that private transitions and inconsistent states are only sound to use in the absence of call/cc. Formally, we disallow them by redefining Island_n as follows:

$$\text{Island}'_n \stackrel{\text{def}}{=} \{\iota \in \text{Island}_n \mid \iota.\varphi = \iota.\delta \wedge \iota.\lightning = \emptyset\}$$

Under this definition, the two notions of world extension coincide and all worlds are consistent. The rest of the model stays the same. In particular, proofs done in the **HOS** model that do not make use of private transitions or inconsistent states can be transferred without any change. The soundness and completeness proofs carry over as well. The former merely needs to be extended in a straightforward way to deal with $\mathsf{call/cc}$, throw, and cont.

7.3 FOS

In the first-order state setting, observe that, for the types of values that can be stored in the heap—namely, those of base type—our logical relation for values coincides with syntactic equality. Consequently, when expressing that two heap values are logically related, we no longer need to refer to a world. Obtaining the model for **FOS** from the one for **HOS** is therefore very simple—all that is needed is to remove the ability of heap relations to be world-dependent:

$$\text{HeapRel}'_n \stackrel{\text{def}}{=} \mathcal{P}(\text{Heap} \times \text{Heap})$$

Our heap relations are now more or less the same as in Pitts and Stark [28]—that is, they are simply heap relations! Correspondingly, we must also update the definitions of $(h_1, h_2) : W, \psi' \otimes \psi''$, and $\mathcal{V}[\![\mathsf{ref}\ \tau]\!]\rho$, all in the obvious manner, to reflect the lack of world indices in heap relations. (For details, see the appendix.) Note that while step-indices are no longer needed to stratify our worlds, they are still useful in modeling general recursive types.

This simplification of HeapRel enables backtracking (see Section 6.1) by isolating islands from one another completely. Whereas before, changing the state of an island ι could break the heap constraints in other islands if we did not strictly follow ι's STS, now there is no way for changes to ι's state to affect the satisfaction of other islands' heap constraints, so we are free to backtrack.

7.4 FOSC

The changes to the **HOS** model discussed in Sections 7.2 and 7.3 are completely orthogonal and may be easily combined in order to obtain a fully abstract model for **FOSC**.

7.5 Proof of Deferred Divergence Example (FOS Version)

We now present in detail a proof that demonstrates the use of all three of our model's special features (private transitions, inconsistent states, and backtracking). Concretely, we show the difficult direction of approximation in the **FOS** version of the "deferred divergence" example from Section 6.2.

Formally, our goal is to prove $\cdot; \cdot; \cdot \vdash e_1 \precsim_{\text{log}} e_2 : \tau$. Unfolding the definition, this reduces to showing $(W, e_1, e_2) \in \mathcal{E}[\![\tau]\!]$ for $W \in$ World. So assume we are given continuations $(W, K_1, K_2) \in \mathcal{K}[\![\tau]\!]$ and heaps $(h_1, h_2) : W$ and $\langle h_1; K_1[e_1]\rangle$ terminates in less than $W.k$ steps. We must now show that W is consistent and that $\langle h_2; K_2[e_2]\rangle$ terminates as well.

Observe that since $\langle h_1; K_1[e_1]\rangle$ terminates in less than $W.k$ steps, so does $\langle h_1 \uplus \{l_y {\mapsto} \mathsf{ff}\}; K_1[\widehat{e_1}[l_y/y]]\rangle$, where $\widehat{e_1}$ is the body of the let-expression in e_1, and l_y is some fresh location. For this new location, we extend the world with an island representing the STS from Section 6.2, with $s = 1$, 2, and 3 representing the left, middle, and right states of the STS, respectively:

$$\begin{array}{rcl} W_s &=& (W.k, (W.\Sigma_1, l_y{:}\mathsf{bool}), W.\Sigma_2, (W.\omega, \iota_s)) \\ \iota_s &=& (s, \delta, \varphi, \lightning, H) \\ \delta &=& \{(1,2),(2,3)\}^* \\ \varphi &=& \{(2,3)\}^* \\ \lightning &=& \{3\} \\ H(1) &=& \{(\widetilde{h_1}, \widetilde{h_2}) \mid \widetilde{h_1}(l_y) = \mathsf{ff}\} \\ H(2) &=& \{(\widetilde{h_1}, \widetilde{h_2}) \mid \widetilde{h_1}(l_y) = \mathsf{ff}\} \\ H(3) &=& \{(\widetilde{h_1}, \widetilde{h_2}) \mid \widetilde{h_1}(l_y) = \mathsf{tt}\} \end{array}$$

Here the superscript "*" in the definitions of δ and φ denotes the reflexive, transitive closure over State.

Note that ι_1 is safe and therefore $W_1 \sqsupseteq^{\mathsf{pub}} W$. Given how we defined our island, it is easy to see that $(h_1 \uplus \{l_y {\mapsto} \mathsf{ff}\}, h_2) : W_1$ follows from $(h_1, h_2) : W$. Assuming we are able to show $(W_1, \widehat{e_1}[l_y/y], e_2) \in \mathcal{V}[\![\tau]\!]$, we can instantiate $(W, K_1, K_2) \in \mathcal{K}[\![\tau]\!]$ and get consistent(W_1) and that $\langle h_2; K_2[e_2]\rangle$ terminates. The latter is one of the two things we needed to show. The other

one is $\mathrm{consistent}(W)$. Since the only difference between W and W_1 is our island, this follows from $\mathrm{consistent}(W_1)$.

It remains to show $(W_1, \widehat{e_1}[l_y/y], e_2) \in \mathcal{V}[\![\tau]\!]$. So suppose we are given a future world $W' \sqsupseteq W_1$ and related callbacks $(W', f_1, f_2) \in \mathcal{V}[\![(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}]\!]$. We need to show $(W', e_1', f_2\ (\lambda_.\perp)) \in \mathcal{E}[\![\mathsf{unit}]\!]$, where

$$e_1' \quad = \quad f_1\ (\lambda_.\, l_y := \mathsf{tt}); \mathsf{if}\ !l_y\ \mathsf{then}\ \perp\ \mathsf{else}\ \langle\rangle.$$

So suppose we are given continuations $(W', K_1', K_2') \in \mathcal{K}[\![\mathsf{unit}]\!]$ and heaps $(h_1', h_2') : W'$ and $\langle h_1'; K_1'[e_1']\rangle$ terminates in less than $W'.k$ steps. We must now show that W' is consistent and that $\langle h_2'; K_2'[f_2\ (\lambda_.\perp)]\rangle$ terminates as well.

As a matter of notation, let W_s' denote the world obtained from W' by setting our island's state to s. We only show the case $W' = W_1'$ here; the other two are similar (and simpler). The first step is to "move to the middle state (state 2)". Formally, since the heap constraints of state 1 and 2 are the same, $(h_1', h_2') : W_1'$ implies $(h_1', h_2') : W_2'$. Now, we want to prove the following:

1. $(W_2', f_1\ (\lambda_.\, l_y := \mathsf{tt}), f_2\ (\lambda_.\perp)) \in \mathcal{E}[\![\mathsf{unit}]\!]$
2. $(W_2', K_1'[\bullet; \mathsf{if}\ !l_y\ \mathsf{then}\ \perp\ \mathsf{else}\ \langle\rangle], K_2') \in \mathcal{K}[\![\mathsf{unit}]\!]$

If we can prove these two subgoals, then instantiating (1) with (2) yields $\mathrm{consistent}(W_2')$ and that $\langle h_2'; K_2'[f_2\ (\lambda_.\perp)]\rangle$ terminates. The latter is one of the two things we needed to show. The other one is $\mathrm{consistent}(W_1')$, which obviously follows from $\mathrm{consistent}(W_2')$. So it remains to show (1) and (2).

For (1), first note that since f_1 and f_2 are related in W_1', they are by monotonicity also related in W_2' since $W_2' \sqsupseteq W_1'$. It therefore suffices to show the relatedness of their thunk arguments, *i.e.,* $(W_2', (\lambda_.\, l_y := \mathsf{tt}), (\lambda_.\perp)) \in \mathcal{V}[\![\mathsf{unit} \to \mathsf{unit}]\!]$. To that end, we suppose $W'' \sqsupseteq W_2'$ and have to show $(W'', l_y := \mathsf{tt}, \perp) \in \mathcal{E}[\![\mathsf{unit}]\!]$. So assume we are given continuations $(W'', K_1'', K_2'') \in \mathcal{K}[\![\mathsf{unit}]\!]$ and heaps $(h_1'', h_2'') : W''$. With the help of the inconsistent state we will now show that $\langle h_1''; K_1''[l_y := \mathsf{tt}]\rangle$ certainly does not terminate in less than $W''.k$ steps (so there is nothing further to do). Assume it does, implying that $\langle h_1''[l_y{\mapsto}\mathsf{tt}]; K_1''[\langle\rangle]\rangle$ does, too. Since $W'' \sqsupseteq W_2'$, W'' is either W_2'' or W_3'' (using the same notational trick as above). Consequently, it is easy to see that $W_3'' \sqsupseteq^{\mathsf{pub}} W''$, as well as $(h_1''[l_y{\mapsto}\mathsf{tt}], h_2'') : W_3''$. Instantiating $(W'', K_1'', K_2'') \in \mathcal{K}[\![\mathsf{unit}]\!]$ with all this plus the trivial fact that $(W_3'', \langle\rangle, \langle\rangle) \in \mathcal{V}[\![\mathsf{unit}]\!]$ yields $\mathrm{consistent}(W_3'')$, which is clearly in contradiction to 3 being an inconsistent state.

For (2), suppose we are given $W'' \sqsupseteq^{\mathsf{pub}} W_2'$ and heaps $(h_1'', h_2'') : W''$ and that $\langle h_1''; K_1'[\mathsf{if}\ !l_y\ \mathsf{then}\ \perp\ \mathsf{else}\ \langle\rangle]\rangle$ terminates in less than $W''.k$ steps. We have to show $\mathrm{consistent}(W'')$ and that $\langle h_2''; K_2'[\langle\rangle]\rangle$ terminates. From the assumptions it is clear that $h_1''(l_y)$ must be ff and thus $\langle h_1''; K_1'[\langle\rangle]\rangle$ terminates in less than $W''.k$ steps. This also implies that W'' must be W_2''. We now want to instantiate $(W_1', K_1', K_2') \in \mathcal{K}[\![\mathsf{unit}]\!]$, but W_2'' does not publicly extend W_1' because there is no public transition from state 1 to state 2. However, we can now *backtrack* to state 1: because both states express the same heap constraint and because heap relations for **FOS** are world-independent, $(h_1'', h_2'') : W_2''$ implies $(h_1'', h_2'') : W_1''$. Note that $W_2'' \sqsupseteq^{\mathsf{pub}} W_2'$ implies $W_1'' \sqsupseteq^{\mathsf{pub}} W_1'$. Finally, we can instantiate $(W_1', K_1', K_2') \in \mathcal{K}[\![\mathsf{unit}]\!]$ with all this plus $(W_1'', \langle\rangle, \langle\rangle) \in \mathcal{V}[\![\mathsf{unit}]\!]$, to obtain $\mathrm{consistent}(W_1'')$ and that $\langle h_2''; K_2'[\langle\rangle]\rangle$ terminates. Since our state 2 is a consistent state, $\mathrm{consistent}(W_1'')$ implies $\mathrm{consistent}(W_2'')$, and we are done.

8. Reasoning in the Presence of Exceptions

In this paper, we have focused attention on first-class continuations as our control effect of interest, and demonstrated that their absence enables the extension of our STS-based Kripke model with the mechanisms of private transitions and inconsistent states. It is natural, then, to ask about the impact that other control effects have on our model. At least in the case of *exceptions*, the answer is quite simple, as we will now briefly explain. (Details appear in the technical appendix [8], and we intend to elaborate on these in an extended version of this paper. We leave consideration of other control effects, such as delimited continuations, to future work.)

First of all, unlike throwing to a continuation, raising an exception causes a "well-bracketed" kind of control effect, in the sense that it passes control to the exception handler that was most recently pushed onto the control stack. Thus, the presence of exceptions does not *per se* restrict our STS model: we are free to use STS's with private transitions and inconsistent states.

However, the possibility of exceptional behavior means that, when proving two *continuations* to be logically related (by $\mathcal{K}[\![\tau]\!]\rho$), we must show that they behave in a related manner not only when they are plugged with related values, but also when they are passed related raised exceptions. Concretely, the definition of $\mathcal{K}[\![\tau]\!]\rho$ becomes the following (assuming a new base type exn of exceptions):

$$\begin{array}{l}\{(W, K_1, K_2) \in \mathrm{ContAtom}[\rho_1(\tau), \rho_2(\tau)] \mid \\ \quad \forall W', v_1, v_2.\, W' \sqsupseteq^{\mathsf{pub}} W \implies \\ \quad ((W', v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho \implies (W', K_1[v_1], K_2[v_2]) \in \mathcal{O}) \wedge \\ \quad ((W', v_1, v_2) \in \mathcal{V}[\![\mathsf{exn}]\!] \implies \\ \qquad (W', K_1[\mathsf{raise}\ v_1], K_2[\mathsf{raise}\ v_2]) \in \mathcal{O})\}\end{array}$$

In essence, this new definition is equivalent to $\mathcal{K}[\![\mathbf{M}(\tau)]\!]\rho$, where $\mathbf{M}$ is the *exception monad—i.e.,* $\mathbf{M}(\tau) \approx \tau + \mathsf{exn}$.

Each of the various examples we have considered in this paper involves proving equivalence of two higher-order functions that, when called, will manipulate some local state and invoke their (unknown) callback arguments. Thus, for each of the examples, the new, more restrictive definition of $\mathcal{K}[\![\tau]\!]\rho$ requires us to consider the possibility that the callback invocation may raise an exception. Since the higher-order function in each example does not install any exception handler around its callback invocation, any exception raised by that callback invocation will remain uncaught, causing the function to return immediately (raising the same exception).

We therefore need to show that any state in which the callback may raise an exception—*i.e.,* any state that is publicly accessible from the one in which the callback was invoked—is also publicly accessible from the initial state in which the higher-order function was called. For the callback-with-lock example, this is indeed the case, since the only state publicly accessible from the "locked" state (in which the callback is invoked) is itself, which is publicly accessible from the "unlocked" starting state. For the other examples, on the other hand, this criterion is not met; and indeed, in the presence of exceptions, it is not hard to find program contexts that distinguish the higher-order functions in those examples.

9. Related and Future Work

Many techniques have been proposed for reasoning about contextual equivalence of stateful programs. Using a variety of these techniques, most of the examples we discuss in this paper *have* been proven already (with minor variations) in different language settings, but there has not heretofore been any clear account of how they all fit together. Indeed, our main contribution lies in our unifying framework of STS's, along with the realization that the absence of call/cc and/or higher-order state enables the extension of our STS model in orthogonal ways. That said, some of our examples are also new, such as "callback with lock" in **FOSC**, and the other ADR examples in **HOSC** (see the appendix [8] for more).

Game Semantics As explained in the introduction, game semantics has served as an inspiration to us, especially Abramsky's idea of the "semantic cube". There are many papers on this topic; perhaps the two most relevant to our present work are Laird's model

of call-by-name PCF extended with a control operator [19] and Abramsky, Honda, and McCusker's model of call-by-value PCF extended with general references [1]. Unlike our **HOSC** and its fragments, the language considered by Abramsky *et al.* does not support pointer equality.[5]

The primary focus of the research on games models has been full abstraction. One of the key motivations for having a fully abstract model is, of course, that it allows one to prove two programs observationally equivalent by proving that their denotations (in games models, "strategies") are the same. However, the games models do not in general directly facilitate such proofs since the strategies are non-trivial to analyze for equality (and since game categories also involve a non-trivial quotienting). Hence, proof methods for proving actual program equivalences based on specific games models have primarily been developed only for simple languages with state, namely call-by-name Idealized Algol. For a finitary version of that language (*i.e.,* a version with only finite ground types and no recursion) there is a full classification of when contextual equivalence is decidable (*e.g.,* see [12, 25]). A finitary version of a call-by-value variant has also been studied by Murawski [24], and with that model he could show some finitary versions of the examples of Pitts and Stark, *e.g.,* the profiling example [24, p. 29].

Another focus of game semantics is on understanding how the presence of different features in a language affects the kinds of interactions a program can have with its context. Laird [19] models the presence of control operators by relaxing the "well-bracketing" restriction on strategies. Abramsky *et al.* [1] model the presence of higher-order state by relaxing the "visibility" restriction. There seems intuitively to be some correspondence between the former and our private transitions, and between the latter and our backtracking, but determining the precise nature of this correspondence is left to future work.

Operational Game Semantics Another line of related work concerns what some have called "operational game semantics". This work considers labeled transition systems, and either traces or bisimulation relations over those, directly inspired by games models. Such so-called "normal form bisimulation" relations have been developed for an untyped language with state and control [34], for a typed language with recursive types (but no state) [21], and for a language with impredicative polymorphism (but no state) [22]. Laird [20] gave a fully abstract trace semantics for the language of Abramsky *et al.* [1] extended with pointer equality. His trace-sets may be viewed as deterministic strategies in the sense of game semantics. Normal form bisimulations have been used to prove contextual equivalence of actual examples, *e.g.,* Støvring and Lassen's proof of correctness [34] for the encoding of call/cc via one-shot continuations that we described at the end of Section 4. Koutavas and Lassen have shown, in unpublished work [16], how Laird's trace semantics can be used to prove the **HOS** version of the deferred divergence example (Section 5.2), by showing that the two programs have the same set of traces.

To the best of our knowledge, however, no fully abstract games model (either operational or denotational) has yet been given for the rich language that we consider in this paper (call-by-value, impredicative polymorphism, general references with pointer equality, call/cc, and recursive types).

Logical Relations Our work is heavily indebted to the pioneering work of Pitts and Stark [28], who gave a fully abstract logical relation for a simply-typed functional language with recursion and first-order state. In particular, we rely on the basic setup of their biorthogonal Kripke model, although (like ADR's) ours is also step-indexed. In the absence of step indices, biorthogonality renders the logical relation *admissible* (an important property when modeling recursion). In the presence of step indices, admissibility is not as important, since the model essentially only consists of finite approximations, and there is no need to ever talk about their limit. Nevertheless, as we have seen, biorthogonality plays a crucial role in modeling control and ensuring full abstraction.

With respect to the latter, it is not clear how useful the full abstraction property is for us *per se*, since it is achieved in a largely "feature-independent" manner. That is, the proof that biorthogonality makes the logical relation complete is essentially the same for each of the four languages we consider, so full abstraction here is perhaps not the most informative criterion. One could for instance take Pitts and Stark's original model, add step-indexing to it, and get out a different fully abstract model for **HOSC**. Clearly, that model would not be as practically powerful as our STS-based model, but it would nevertheless be fully abstract.

Aside from ADR, the closest logical relations to ours are the ones developed by Bohr in her thesis [7]. Hers also employ biorthogonality, albeit in a denotational setting. Her possible worlds bear some similarity to ADR's in that they, too, allow one to model heap properties that evolve over time. In addition, they allow one to impose constraints on continuations. Like us, she is also able to handle the **HOS** version of the deferred divergence example, but the language she considers is not as rich as ours (it does not support full polymorphism), and she does not consider handling call/cc or the restriction to first-order state. We can prove all of the examples from her thesis, and we believe that our proofs are significantly simpler to understand.

Regarding the deferred divergence example: it is originally due to O'Hearn, who formulated it in the context of Idealized Algol [26, 2.3]. Pitts showed how to prove this example using operational Kripke logical relations, by allowing the parameters of the logical relation to relate proper states to undefined states (*i.e.,* by phrasing heap relations over "lifted" heaps) [29]. It is not clear whether this technique generalizes to higher-order state, however.

More recently, Johann, Simpson, and Voigtländer [14] have proposed a generic framework for operational reasoning about algebraic effects. Their work is complementary to ours: they develop effect-independent proof principles, whereas we develop effect-specific proof principles. They do not consider local state, higher-order state, or control.

Lastly, our decision to employ both step-indexing and biorthogonality was influenced directly by the work of Benton, together with Tabareau [6] and Hur [5], on compiler correctness. They argue persuasively for the benefits of combining the two techniques.

Environmental Bisimulations For reasoning about contextual equivalences (involving either type abstraction or local state), one of the most successful alternatives to logical relations is the coinductive technique of *environmental bisimulations*. The current state of the art is Sumii's work on type abstraction and general references [35], which builds on work by Sumii-Pierce [36], Koutavas-Wand [17], and Sangiorgi-Kobayashi-Sumii [32]. Sumii is able to handle all the examples we have presented here in the setting of **HOS**; he does not consider call/cc or first-order state (but does, in the work with Sangiorgi, consider concurrency). In some cases (*e.g.,* for the well-bracketed version of the "awkward" example—see Section 5.1), his approach is somewhat "brute-force" in the sense that it requires explicit reasoning about the intensional structure of program contexts. We believe our state transition systems capture the intuitions about well-bracketing at a more abstract level.

Anti-Frame Rule Pottier [30] has proposed an alternative way of reasoning about local state using a rich type system with capabilities, regions, and linearity. His *anti-frame rule* allows one to es-

[5] We have not emphasized the fact that we model pointer equality in this paper, but some of ADR's examples do make use of it, and it is a feature one generally expects to find in real ML-like languages.

tablish a hidden property about a piece of local state, much in the same way that our islands do. In its original form, however, the anti-frame rule was restricted to reasoning about *invariants*, which we argued in Section 3 are insufficient for many examples.

To address this limitation, Pottier has suggested two extensions of his framework. First, in joint work with Pilkiewicz [27], he proposes the use of *fates*, which enable reasoning about *monotonic* state in a manner rather similar to the state transition systems in our Kripke model. Second, in a brief unpublished note [31], he sets forth a *generalized* version of the anti-frame rule that permits reasoning about well-bracketed state change.

While there are clear analogies between these extensions and our public/private state transitions, determining a precise formal correspondence is likely to be difficult because the methods are tailored to different purposes. On one hand, Pottier's type systems are richer than that of ML, and thus his techniques can be used to verify correctness of some interesting programs that exploit the advanced features of his type systems. On the other hand, some equivalences—like our "deferred divergence" example from Section 5.2—do not seem to be easily expressible as "unary" type-checking problems and thus cannot seemingly be handled by Pottier's method. Moreover, like Sumii [35], Pottier restricts attention to languages that support higher-order state but no control effects.

Finally, it is important to note that Pottier's anti-frame rule has only been proven sound in a relatively idealized setting [33], and its soundness has yet to be established even in the context of the type-and-capability system in which it was originally proposed [30], let alone the extended systems mentioned above [27, 31].

Other Related Work Seminal work on operational reasoning about state and control was conducted by Felleisen and Hieb [10] and Mason and Talcott [23], but the proof principles they developed are relatively weak in comparison to the ones afforded by our model. Thielecke [37] demonstrated an interesting equivalence that holds in the presence of exceptions and state, but not in the presence of continuations and state. His proof method is relatively brute-force, however, and we can easily prove his example using an STS with private transitions. More recently, Yoshida *et al.* [38] proposed a Hoare-style logic for reasoning about higher-order programs with local state, but it does not handle abstract types, nor does it permit the kind of reasoning achieved by our STS's. Dreyer *et al.* [9] have devised a relational modal logic that accounts for the essential aspects of the ADR model. In the future, we hope to generalize that logic to account for the additional features we have proposed here.

References

[1] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS*, 1998.

[2] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

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

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

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

[6] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, 2009.

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

[8] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning (Technical appendix), 2010. `http://www.mpi-sws.org/~dreyer/papers/stslr/`.

[9] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[10] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2):235–271, 1992.

[11] D. Friedman and C. Haynes. Constraining control. In *POPL*, 1985.

[12] D. R. Ghica and G. McCusker. Reasoning about Idealized Algol using regular languages. In *ICALP*, 2000.

[13] P. Johann. Short cut fusion is correct. *JFP*, 13(4):797–814, 2003.

[14] P. Johann, A. Simpson, and J. Voigtländer. A generic operational metatheory for algebraic effects. In *LICS*, 2010.

[15] P. Johann and J. Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

[16] V. Koutavas and S. Lassen. Fun with fully abstract operational game semantics for general references. Unpublished, Feb. 2008.

[17] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, 2006.

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

[19] J. Laird. Full abstraction for functional languages with control. In *LICS*, 1997.

[20] J. Laird. A fully abstract trace semantics for general references. In *ICALP*, 2007.

[21] S. B. Lassen and P. B. Levy. Typed normal form bisimulation. In *CSL*, 2007.

[22] S. B. Lassen and P. B. Levy. Typed normal form bisimulation for parametric polymorphism. In *LICS*, 2008.

[23] I. Mason and C. Talcott. Equivalence in functional languages with effects. *JFP*, 1(3):287–327, 1991.

[24] A. S. Murawski. Functions with local state: regularity and undecidability. *TCS*, 338(1–3):315–349, 2005.

[25] A. S. Murawski and I. Walukiewicz. Third-order Idealized Algol with iteration is decidable. *TCS*, 390(2–3):214–229, 2008.

[26] P. O'Hearn and U. Reddy. Objects, interference, and the Yoneda embedding. In *MFPS*, 1995.

[27] A. Pilkiewicz and F. Pottier. The essence of monotonic state. Submitted for publication, 2009.

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

[29] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *LICS*, 1996.

[30] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, 2008.

[31] F. Pottier. Generalizing the higher-order frame and anti-frame rules. Unpublished, 2009.

[32] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *LICS*, 2007.

[33] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus. A semantic foundation for hidden state. In *FOSSACS*, 2010.

[34] K. Støvring and S. B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*, 2007.

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

[36] E. Sumii and B. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):1–43, 2007.

[37] H. Thielecke. On exceptions versus continuations in the presence of state. In *ESOP*, 2000.

[38] N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. *LMCS*, 4(4:2), 2008.