State-Dependent Representation Independence

Amal Ahmed
TTI-C
amal@tti-c.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.mpg.de

Andreas Rossberg
MPI-SWS
rossberg@mpi-sws.mpg.de

Abstract

Mitchell's notion of *representation independence* is a particularly useful application of Reynolds' relational parametricity — two different implementations of an abstract data type can be shown contextually equivalent so long as there exists a relation between their type representations that is preserved by their operations. There have been a number of methods proposed for proving representation independence in various pure extensions of System F (where data abstraction is achieved through existential typing), as well as in Algol- or Java-like languages (where data abstraction is achieved through the use of local mutable state). However, none of these approaches addresses the *interaction* of existential type abstraction and local state. In particular, none allows one to prove representation independence results for *generative* ADTs — *i.e.*, ADTs that *both* maintain some local state *and* define abstract types whose internal representations are dependent on that local state.

In this paper, we present a syntactic, logical-relations-based method for proving representation independence of generative ADTs in a language supporting polymorphic types, existential types, general recursive types, and unrestricted ML-style mutable references. We demonstrate the effectiveness of our method by using it to prove several interesting contextual equivalences that involve a close interaction between existential typing and local state, as well as some well-known equivalences from the literature (such as Pitts and Stark's "awkward" example) that have caused trouble for previous logical-relations-based methods.

The success of our method relies on two key technical innovations. First, in order to handle generative ADTs, we develop a possible-worlds model in which relational interpretations of types are allowed to *grow* over time in a manner that is tightly coupled with changes to some local state. Second, we employ a *step-indexed* stratification of possible worlds, which facilitates a simplified account of mutable references of higher type.

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

General Terms Languages, Theory, Verification

Keywords Abstract data types, representation independence, existential types, local state, step-indexed logical relations

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

1. Introduction

Reynolds' notion of *relational parametricity* [23] is the essence of type abstraction — clients of an abstract type behave uniformly across *all* relational interpretations of that type and thus cannot depend in any way on how the type is represented. Mitchell's notion of *representation independence* [17] is a particularly useful application of relational parametricity — two different implementations of an abstract data type can be shown contextually equivalent so long as there *exists* a relation between their type representations that is preserved by their operations. This is useful even when the type representations of the two ADTs are the same, because the choice of an arbitrary relational interpretation for the abstract type allows one to establish the existence of local invariants.

Originally these ideas were developed in the context of (variants of) System F, but over the past two decades there has been a great deal of work on extending them to the setting of more realistic languages, such as those with recursive functions [20], general recursive types [16, 1, 11], selective strictness [29], etc. In these *functional* languages, data abstraction is achieved through the use of existential types. Others have considered representation independence in the setting of *imperative* languages, such as Algol and Java, where data abstraction is achieved instead through the use of local mutable state (*e.g.*, local variables or private fields) [21, 5, 14].

Of course, most modern languages (such as ML) are neither purely functional nor imperative, but rather freely mix the paradigms. However, none of the existing work on representation independence has considered a language supporting both the functional and the imperative approaches to data abstraction, *i.e.*, both existential types and local state. This is unfortunate, since both abstraction mechanisms play important, interdependent roles in the definition of *generative* abstract data types.

1.1 Reasoning About Generative Abstract Data Types

Existential type abstraction provides *type generativity* — every unpacking of an existential package generates a *fresh* abstract type that is distinct from any other. This is similar to the behavior of Standard ML's *generative functors*, which generate fresh abstract types at each application, and indeed the semantics of SML-style functors may be understood as a stylized use of existential type abstraction [25]. The clearest motivation for type generativity is in the definition of ADTs that encapsulate some local state. In such instances, generativity is sometimes *necessary* to achieve the proper degree of data abstraction.

As a simple motivating example, consider the SML module code in Figure 1, which is adapted from an example of Dreyer *et al.* [12]. (Later in the paper, we will develop a similar example using existential types.) Here, the signature `SYMBOL` describes a module implementing a mutable symbol table, which maps "symbols" to strings. The module provides an abstract type `t` describing the symbols currently in its table; a function `eq` for comparing symbols for equality; a function `insert`, which adds a

```
signature SYMBOL = sig
  type t
  val eq : t * t -> bool
  val insert : string -> t
  val lookup : t -> string
end
functor Symbol () :> SYMBOL = struct
  type t = int
  val size = ref 0
  val table = ref nil
  fun eq (x,y) = x = y
  fun insert str = (
    size := !size + 1;
    table := str :: !table;
    !size
  )
  fun lookup n =
    List.nth (!table, !size - n)
end
```

Figure 1. Generativity Example

given string to the table and returns a fresh symbol mapped to it; and a function `lookup`, which looks up a given symbol in the table and returns the corresponding string.

The functor `Symbol` implements the symbol type `t` as an integer index into a (mutable) list of strings. When applied, `Symbol` creates a fresh `table` (represented as a pointer to an empty list) and a mutable counter `size` (representing the size of the table). The implementations of the various functions are straightforward, and the body of the functor is sealed with the signature `SYMBOL`, thus hiding access to the local state (`table` and `size`).

The call to `List.nth` in the `lookup` function might in general raise a `Subscript` exception if the input `n` were an arbitrary integer. However, we "know" that this cannot happen because `lookup` is exported with argument type `t`, and the only values of type `t` that a client could possibly have gotten hold of are the values returned by `insert`, *i.e.*, integers that are between 1 and the current size of `table`. Therefore, the implementation of the `lookup` function need not bother handling the `Subscript` exception.

This kind of reasoning is commonplace in modules that encapsulate local state. But what justifies it? Intuitively, the answer is type generativity. Each instantiation of the `Symbol` functor creates a fresh symbol type `t`, which represents the type of symbols that are valid in its own `table` (but not any other). Were `Symbol` not generative, each application of the `Symbol` functor would produce a module with distinct local state but the *same* symbol type. It would then be easy to induce a `Subscript` error by accidentally passing a value of one table's symbol type to another's `lookup` function.[1]

While this intuition about the importance of generativity is very appealing, it is also completely informal. The goal of this paper is to develop a formal framework for reasoning about the interaction of generative type abstraction and mutable state.

In the case of an example like the `Symbol` functor, we will be able to show that the implementation of `Symbol` shown in Figure 1 is *contextually equivalent* to one whose `lookup` function is replaced by:

```
fun lookup n =
  if n > 0 andalso n <= !size
           andalso !size = length(!table)
  then List.nth (!table, !size - n)
  else "Hell freezes over"
```

[1] This is the case, for example, in OCaml, which only supports *applicative* (*i.e.*, non-generative) functors [15].

In other words, there is no observable difference between the original `Symbol` functor and one that dynamically checks the various invariants we claim to "know." Hence, the checks are unnecessary.

This kind of result can be understood as an instance of representation independence, albeit a somewhat degenerate one in that the ADTs in question share the same type representation. As with most such results, the proof hinges on the construction of an appropriate relational interpretation of the abstract type `t`, which serves to impose an invariant on the possible values of type `t`. In this case, we wish to assert that for a particular structure `S` defined by `Symbol()`, the only values of type `S.t` are integers between 1 and the current size of `S`'s `table`. This will allow us to prove that any range check on the argument to `S`'s `lookup` function is superfluous.

The problem is that the relational interpretation we wish to assign to `S.t` depends on the *current* values stored in `S`'s local state. In effect, as `S`'s `insert` function is called repeatedly over time, its `table` grows larger, and the relational interpretation of `S.t` must grow accordingly to include more and more integers. Thus, what we need is an account of *state-dependent representation independence*, in which the relational interpretations of abstract types are permitted to *grow* over time, in a manner that is tightly coupled with changes to some local state.

1.2 Overview

In this paper, we present a novel method for proving state-dependent representation independence results. Our method extends previous work by Ahmed on syntactic *step-indexed* logical relations for recursive and quantified types [1]. We extend her technique with support for reasoning about local state, and demonstrate its effectiveness on a variety of small but representative examples. Although our primary focus is on proving representation independence for ADTs that exhibit an interaction of existentials and state, our method also handles several well-known simply-typed examples from the literature on local state (such as Pitts and Stark's "awkward" example [21]) that have proven difficult for previous logical-relations-based methods to handle.

In order to reason about local state, we build into our logical relation a notion of *possible worlds*. While several aspects of our possible worlds are derived from and inspired by prior work, other aspects are quite novel:

1. We enrich our possible worlds with *populations* and *laws*, which allow us to evolve the relational interpretation of an abstract type over time in a controlled, state-dependent fashion. For instance, we can use a population to grow a set of values (*e.g.*, the integers between 1 and some n), together with a law that explains what the current population implies about the current machine state (*e.g.*, that the symbol table has size n).
2. Second, our method provides the ability to reason locally about references to higher-order values. While ours is not the first method to handle higher-order state, our approach is novel and arguably simpler than previous accounts. It depends critically on step-indexing in order to avoid a circularity in the construction of possible worlds.

The remainder of the paper is structured as follows. In Section 2, we present $\mathsf{F}^{\mu!}$, our language under consideration, which is essentially System F extended with general recursive types and general ML-style references. In Section 3, we explain at a high level how our method works and what is novel about it. In Section 4, we present the details of our logical relation and prove it sound (but not complete) with respect to contextual equivalence of $\mathsf{F}^{\mu!}$ programs. In Section 5, we show how to use our method to prove a number of interesting contextual equivalences. Finally, in Section 6, we conclude with a thorough comparison to related work, as well as directions for future work.

$$
\begin{array}{lll}
\text{Types} & \tau & ::= \alpha \mid \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \\
& & \quad \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mu\alpha.\,\tau \mid \mathsf{ref}\,\tau \\
\text{Prim Ops} & o & ::= + \mid - \mid = \mid < \mid \leq \mid \ldots \\
\text{Terms} & e & ::= x \mid () \mid l \mid \pm n \mid o(e_1, \ldots, e_n) \mid \mathtt{true} \mid \mathtt{false} \mid \\
& & \quad \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \mid \langle e_1, e_2 \rangle \mid \mathtt{fst}\, e \mid \mathtt{snd}\, e \mid \\
& & \quad \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha.\, e \mid e\,[\tau] \mid \\
& & \quad \mathtt{pack}\,\tau, e\,\mathtt{as}\,\exists\alpha.\,\tau' \mid \mathtt{unpack}\, e_1\,\mathtt{as}\,\alpha, x\,\mathtt{in}\, e_2 \mid \\
& & \quad \mathtt{fold}\, e \mid \mathtt{unfold}\, e \mid \mathtt{ref}\, e \mid \mathtt{!}e \mid e_1 \mathtt{:=} e_2 \mid e_1 \mathtt{==} e_2 \\
\text{Values} & v & ::= () \mid l \mid \pm n \mid \mathtt{true} \mid \mathtt{false} \mid \langle v_1, v_2 \rangle \mid \\
& & \quad \lambda x\!:\!\tau.\, e \mid \Lambda\alpha.\, e \mid \mathtt{pack}\,\tau_1, v\,\mathtt{as}\,\exists\alpha.\,\tau \mid \mathtt{fold}\, v
\end{array}
$$

$$
\begin{array}{rcll}
s, (\lambda x\!:\!\tau.\, e)\, v & \longmapsto & s, [v/x]e & \\
s, (\Lambda\alpha.\, e)\,[\tau] & \longmapsto & s, [\tau/\alpha]e & \\
s, \mathtt{unpack}\,(\mathtt{pack}\,\tau, v\,\mathtt{as}\,\exists\alpha.\,\tau')\,\mathtt{as}\,\alpha, x\,\mathtt{in}\, e & & & \\
\qquad & \longmapsto & s, [\tau/\alpha][v/x]e & \\
s, \mathtt{unfold}\,(\mathtt{fold}\, v) & \longmapsto & s, v & \\
s, \mathtt{ref}\, v & \longmapsto & s[l \mapsto v], l & \text{where } l \notin \mathrm{dom}(s) \\
s, \mathtt{!}l & \longmapsto & s, v & \text{where } s(l) = v \\
s, l\mathtt{:=}v & \longmapsto & s[l \mapsto v], () & \text{where } l \in \mathrm{dom}(s) \\
s, l\mathtt{==}l & \longmapsto & s, \mathtt{true} & \\
s, l\mathtt{==}l' & \longmapsto & s, \mathtt{false} & \text{where } l \neq l'
\end{array}
$$

$$
\frac{s, e \longmapsto s', e'}{s, E[e] \longmapsto s', E[e']}
$$

$$
\begin{array}{lll}
\text{Type Contexts} & \Delta & ::= \cdot \mid \Delta, \alpha \\
\text{Value Contexts} & \Gamma & ::= \cdot \mid \Gamma, x : \tau \\
\text{Store Typing} & \Sigma & ::= \cdot \mid \Sigma, l : \tau \qquad \text{where } FTV(\tau) = \emptyset
\end{array}
$$

$$
\frac{\Gamma(x) = \tau}{\Delta;\Gamma;\Sigma \vdash x : \tau} \quad \cdots \quad \frac{\Sigma(l) = \tau}{\Delta;\Gamma;\Sigma \vdash l : \mathsf{ref}\,\tau} \quad \frac{\Delta;\Gamma;\Sigma \vdash e : \tau}{\Delta;\Gamma;\Sigma \vdash \mathtt{ref}\, e : \mathsf{ref}\,\tau}
$$

$$
\frac{\Delta;\Gamma;\Sigma \vdash e : \mathsf{ref}\,\tau}{\Delta;\Gamma;\Sigma \vdash \mathtt{!}e : \tau} \qquad \frac{\Delta;\Gamma;\Sigma \vdash e_1 : \mathsf{ref}\,\tau \quad \Delta;\Gamma;\Sigma \vdash e_2 : \tau}{\Delta;\Gamma;\Sigma \vdash e_1 \mathtt{:=} e_2 : \mathsf{unit}}
$$

$$
\frac{\Delta;\Gamma;\Sigma \vdash e_1 : \mathsf{ref}\,\tau \quad \Delta;\Gamma;\Sigma \vdash e_2 : \mathsf{ref}\,\tau}{\Delta;\Gamma;\Sigma \vdash e_1 \mathtt{==} e_2 : \mathsf{bool}}
$$

Well-typed Store:

$$
\frac{\forall l \in \mathrm{dom}(\Sigma).\ \cdot;\cdot;\Sigma \vdash s(l) : \Sigma(l)}{\vdash s : \Sigma}
$$

Figure 2. $\mathsf{F}^{\mu!}$ Syntax + Dynamic and Static Semantics (excerpts)

2. The Language $\mathsf{F}^{\mu!}$

We consider $\mathsf{F}^{\mu!}$, a call-by-value λ-calculus with impredicative polymorphism, iso-recursive types, and general ML-style references. The syntax of $\mathsf{F}^{\mu!}$ is shown in Figure 2, together with excerpts of the static and dynamic semantics. We assume an infinite set of locations Loc ranged over by l. Our term language includes equality on references ($e_1 \mathtt{==} e_2$), but is otherwise standard.

We define a small-step operational semantics for $\mathsf{F}^{\mu!}$ as a relation between configurations (s, e), where s is a global store mapping locations l to values v. We use evaluation contexts E to lift the primitive reductions to a standard left-to-right call-by-value semantics for the language. We elide the syntax of evaluation contexts as it is completely standard, and we show only some of the reduction rules in Figure 2.

$\mathsf{F}^{\mu!}$ typing judgments have the form $\Delta;\Gamma;\Sigma \vdash e : \tau$ where the contexts Δ, Γ, and Σ are defined as in Figure 2. The type context Δ is used to track the set of type variables in scope; the value context Γ is used to track the term variables in scope (along with their types τ, which must be well formed in context Δ, written $\Delta \vdash \tau$); and the store typing Σ tracks the types of the contents of locations in the store. Note that Σ maps locations to closed types. We write $FTV(\tau)$ to denote the set of type variables that appear free in type τ. The typing rules are entirely standard, so we show only a few rules in Figure 2. We refer the reader to the online technical appendix [3] for full details of $\mathsf{F}^{\mu!}$.

2.1 Contextual Equivalence

A context C is an expression with a single hole $[\cdot]$ in it. Typing judgments for contexts have the form $\vdash C : (\Delta;\Gamma;\Sigma \vdash \tau) \Rightarrow (\Delta';\Gamma';\Sigma' \vdash \tau')$, where $(\Delta;\Gamma;\Sigma \vdash \tau)$ indicates the type of the hole. Essentially, this judgment says that if e is an expression such that $\Delta;\Gamma;\Sigma \vdash e : \tau$, then $\Delta';\Gamma';\Sigma' \vdash C[e] : \tau'$. The typing rule for a hole $[\cdot]$ is as follows:

$$
\frac{\Delta \subseteq \Delta' \quad \Gamma \subseteq \Gamma' \quad \Sigma \subseteq \Sigma'}{\vdash [\cdot] : (\Delta;\Gamma;\Sigma \vdash \tau) \Rightarrow (\Delta';\Gamma';\Sigma' \vdash \tau)}
$$

The other rules are straightforward (see our online appendix [3]).

We define contextual approximation ($\Delta;\Gamma;\Sigma \vdash e_1 \preceq^{ctx} e_2 : \tau$) to mean that, for any well-typed program context C with a hole of the type of e_1 and e_2, the termination of $C[e_1]$ implies the termination of $C[e_2]$. Contextual equivalence ($\Delta;\Gamma;\Sigma \vdash e_1 \approx^{ctx} e_2 : \tau$) is then defined as approximation in both directions.

Definition 2.1 (Contextual Approximation & Equivalence)

Let $\Delta;\Gamma;\Sigma \vdash e_1 : \tau$ *and* $\Delta;\Gamma;\Sigma \vdash e_2 : \tau$.

$$
\begin{array}{l}
\Delta;\Gamma;\Sigma \vdash e_1 \preceq^{ctx} e_2 : \tau \stackrel{\text{def}}{=} \\
\quad \forall C, \Sigma', \tau', s.\ \vdash C : (\Delta;\Gamma;\Sigma \vdash \tau) \Rightarrow (\cdot;\cdot;\Sigma' \vdash \tau') \ \wedge\ \vdash s : \Sigma' \ \wedge \\
\qquad\qquad s, C[e_1] \Downarrow \implies s, C[e_2] \Downarrow \\
\Delta;\Gamma;\Sigma \vdash e_1 \approx^{ctx} e_2 : \tau \stackrel{\text{def}}{=} \\
\quad \Delta;\Gamma;\Sigma \vdash e_1 \preceq^{ctx} e_2 : \tau \ \wedge\ \Delta;\Gamma;\Sigma \vdash e_2 \preceq^{ctx} e_1 : \tau
\end{array}
$$

3. The Main Ideas

In this section we give an informal overview of the main novel ideas in our method, and how it compares to some previous approaches.

3.1 Logical Relations

Broadly characterized, our approach is a *logical relations* method. We define a relation $\mathcal{V}\,[\![\tau]\!]\,\rho$, which relates pairs of values at type τ, where the free type variables of τ are given relational interpretations in ρ. The relation is "logical" in the sense that its definition follows the structure of τ, modeling each type constructor as a logical connective. For example, arrow types correspond to implication, so functions are defined to be related at an arrow type iff relatedness of their arguments implies relatedness of their results. We will show that this logical relation is sound with respect to contextual equivalence for $\mathsf{F}^{\mu!}$. This is useful because, for many examples, it is much easier to show two programs are in the logical relation than to show they are contextually equivalent directly.

Logical relations methods are among the oldest techniques for proving representation independence results. We will assume the reader is generally familiar with the flavor of these techniques, and instead focus on what is distinctive and original about ours.

3.2 Local Reasoning via Possible Worlds and Islands

As explained in Section 1, our core contribution is the idea of *state-dependent* relational interpretations of abstract types. That is, whether two values are related by some abstract type's relational interpretation may depend on the current state of the heap. But when defining such a relational interpretation, how can we characterize the "current state of the heap?"

As a starting point, we review the general approach taken by a number of prior works on reasoning about local state [21, 22, 7, 10]. This approach, which utilizes a *possible worlds* model, has influenced us greatly, and constitutes the foundation of our method. However, the form it has taken in prior work is insufficient for our purposes, and it is instructive to see why.

The general approach of these prior works is to index the logical relation not only by a type τ but by a *world* W. Instead of characterizing the current state of the heap, W characterizes the properties we expect the heap to have. In other words, it is a relation on machine stores, and we restrict attention to pairs of stores that satisfy it. If two values v_1 and v_2 are in the logical relation at type τ and world W, then it means they are related when considered under any two stores s_1 and s_2, respectively, that satisfy W.

Worlds in turn are constructed as a separating conjunction of smaller worlds $\langle w_1, \ldots, w_n \rangle$, sometimes called *islands*, where each island is a relation that "concerns" a disjoint piece of the store. Intuitively, this means that each island distinguishes between pairs of stores only on the basis of a particular set of memory locations, and the set of locations that one island cares about is disjoint from the set that any other one cares about.

Exactly how the separation criterion on islands is formalized is immaterial; the important point is that it enables local reasoning. Suppose we want to prove that one expression is related to another in world W. Each may allocate some fresh piece of the store, and before showing that the resulting values of the expressions are related, we are permitted to extend W with a new island w describing how these fresh pieces of the store relate to each other. World extension is sound here precisely because the new island is (due to freshness of allocation) separate from the others. So long as the expressions in question do not make the locations in their local stores publicly accessible, no other part of the program is capable of mutating the store in such a manner as to violate w.

To make things concrete and to observe the limitations of possible worlds (at least as we have described them), let us consider the motivating example from Section 1. To prove that the two implementations of the `Symbol` functor are contextually equivalent, we will show that their bodies are logically related in an arbitrary initial world W_0. Both functors allocate local state in the same way, namely by allocating one pointer for `table` and one for `size`, so we will want to extend W_0 with an island w_{sym} describing the local invariants on `table` and `size`. How should we define w_{sym}?

One useful invariant that w_{sym} can enforce is that, for both implementations of `Symbol`, the integer pointed to by `size` is equal to the length of the list pointed to by `table`. By incorporating this property into w_{sym}, we will be guaranteed that, in any *future world* (*i.e.*, any extension of $W_0 \uplus w_{\mathsf{sym}}$) in which the `lookup` function is called, the dynamic check `!size = length(!table)` in the second implementation of `Symbol` will always evaluate to `true`.

We can also use w_{sym} to enforce that `!size` is the same in the stores of both programs, and similarly for `!table`. Unfortunately, while this is a necessary condition, it is not sufficient to prove that the range check on the argument of `lookup` in the second `Symbol` implementation always evaluates to `true`. For that, we need a way of correlating the value of `!size` and the possible values of type `t`, but the islands we have developed thus far do not provide one.

3.3 Populating the Islands and Enforcing the Laws

The problem with islands is that they are static entities with no potential for development. To address this limitation, we enrich islands with *populations*. A population is a set of values that "inhabit" an island and affect the definition of the store relation for that island. An island's population may *grow* over time (*i.e.*, as we move to future worlds), and its store relation may change accordingly. In order to control population growth, we equip every island with an immutable *law* governing the connection between its population and its store relation. We denote populations by V, store relations by ψ, and laws by $\mathcal{L}$.

Consider the `Symbol` example. Let us define $V_n = \{1, \ldots, n\}$, and let ψ_n be the store relation containing pairs of stores that obey the properties concerning `table` and `size` described in Section 3.2 and that, in addition, both map the location `size` to n. The idea is that V_n describes the set of values of type `t`, *given* that the current stores satisfy ψ_n. Thus, when we extend the initial world W_0 with an island w_{sym} governing `Symbol`'s local state, we will choose that w_{sym} to comprise population V_0, store relation ψ_0, and a law $\mathcal{L}$ defined as $\{(\psi_n, V_n) \mid n \geq 0\}$. Here, V_0 and ψ_0 characterize the initial world, in which there are no values of type `t` and the size of the table is 0. The law $\mathcal{L}$ describes what future populations and store relations on this island may look like. In particular, $\mathcal{L}$ enforces that future populations may contain 1 to n (for any n), but only in conjunction with stores that map `size` to n. (Of course, the initial population and store relation must also obey the law, which they do.) An island's law is established when the island is first added to the world and may not be amended in future worlds.

Having extended the world W_0 with this new island w_{sym}, we are now able to define a relational interpretation for the type `t`, namely: values v_1 and v_2 are related at type `t` in world W if $v_1 = v_2 = m$, where m belongs to the population of w_{sym} in W. In proving equivalence of the two versions of the `lookup` function, we can assume that we start with stores s_1 and s_2 that are related by some world W, where W is a future world of $W_0 \uplus w_{\mathsf{sym}}$, and that the arguments to `lookup` are related at type `t` in W. Consequently, given the law that we established for w_{sym} together with the interpretation of `t`, we know that the arguments to `lookup` must both equal some m, that the current population of w_{sym} must be some V_n, where $1 \leq m \leq n$, and that the current store relation must be ψ_n. Since s_1 and s_2 satisfy W, they must satisfy ψ_n, which means they map `size` to $n \geq m$. Hence, the dynamic range check in the second version of `Symbol` must evaluate to `true`.

For the above relational interpretation of `t` to make sense, we clearly need to be able to refer to a particular island in a world (*e.g.*, w_{sym}) by some unique identifier that works in all future worlds. We achieve this by insisting that a world be an ordered list of islands, and that new islands only be added to the end of the list. This allows us to access islands by their position in the list, which stays the same in future worlds.

In addition, an important property of the logical relation, which relational interpretations of abstract types must thus obey as well, is closure under world extension, *i.e.*, that if two values are related in world W, then they are related in any future world of W. To ensure closure under world extension for relations that depend on their constituents' inhabitation of a particular island (such as the relation used above to interpret `t`), we require that island populations can only grow larger in future worlds, not smaller.

For expository purposes, we have motivated our population technique with an example that is deliberately simple, in the sense that the relational interpretation of `t` is completely dependent on the current local state. That is, if we know that the current value of `!size` is n, then we know without a doubt that the relational interpretation of `t` in the current world must be $\{(1, 1), \ldots, (n, n)\}$. In Section 5, we will see more complex examples in which the relational interpretation of `t` may depend not only on the current state, but also on the history of the program execution up to that point. Our population techniques scales very nicely to handle such examples because it allows us to control the *evolution* of an abstract type's relational interpretation over time.

3.4 Mutable References to Higher-Order Values

Most prior possible-worlds logical-relation approaches to reasoning about local state impose serious restrictions on what can be stored in memory. Pitts and Stark [21], for example, only allow references to integers. Reddy and Yang [22] and Benton and Leperchey [7] additionally allow references to *data*, which include integers and pointers but not functions. In the present work, however, we would like to avoid any restrictions on the store.

To see what (we think) is tricky about handling references to higher-order values, suppose we have two programs that both maintain some local state, and we are trying to prove these programs equivalent. Say the invariant on this local state, which we will enforce using an island w, is very simple: the value that the first program stores in location l_1 is logically related to the value that the second program stores in l_2. If these values were just integers, we could write the law for w (as we did in the `Symbol` example) so that in any future world, w's store relation ψ must demand that two stores s_1 and s_2 are related only if $s_1(l_1) = s_2(l_2)$. This works because at type `int`, the logical relation coincides with equality.

If the locations have some higher type τ, however, the definition of w's store relation ψ will need to relate $s_1(l_1)$ and $s_2(l_2)$ using the *logical relation* at type τ, not mere syntactic equality. But the problem is: logical relations are indexed by worlds. In order for ψ to say that $s_1(l_1)$ and $s_2(l_2)$ are related at type τ, it needs to specify the world W in which their relation is being considered.

Bohr and Birkedal [10] address this issue by imposing a rigid structure on their store relations. Specifically, instead of having a single store relation per island, they employ a *local parameter*, which is roughly a set of pairs of the form (P, LL), where P is a store relation and LL is a finite set of pairs of locations (together with a closed type). The way to interpret this local parameter is that the current stores must satisfy *one* of the P's, and all the pairs of locations in the corresponding LL must be related by the logical relation in the *current* world. In the case of our example with l_1 and l_2, they would define a local parameter $\{(P, LL)\}$, where P is the universal store relation and $LL = \{(l_1, l_2, \tau)\}$. Bohr and Birkedal's approach effectively uses the LL's to abstract away explicit references to the world-indexed logical relation within the store relation. This avoids the need to refer to a specific world inside a store relation, but it only works for store relations that are expressible in the highly stylized form of these local parameters.

Instead, our approach is to *parameterize* store relations over the world in which they will be considered. Then, in defining what it means for two stores s_1 and s_2 to satisfy some world W, we require that for every ψ in W, $(s_1, s_2) \in \psi[W]$, *i.e.,* s_1 and s_2 obey ψ when it is instantiated to the current world W. The astute reader will have noticed, however, that this parameterization introduces a circularity: worlds are defined to be collections of store relations, which are now parameterized by worlds. To break this circularity, we employ *step-indexed* logical relations.

3.5 Step-Indexed Logical Relations and Possible Worlds

Appel and McAllester [4] introduced the step-indexed model as a way to express *semantic* type soundness proofs for languages with general recursive and polymorphic types. Although its original motivation was tied to foundational proof-carrying code, the technique has proven useful in a variety of applications. In particular, Ahmed [1] has used a binary version of Appel and McAllester's model for relational reasoning about System F extended with general recursive types, and it is her work that we build on.

The basic idea is closely related to classic constructions from domain theory. We define the logical relation $\mathcal{V}[\![\tau]\!]\rho$ as the limit of an infinite chain of approximation relations $\mathcal{V}_n[\![\tau]\!]\rho$, where $n \geq 0$. Informally, values v_1 and v_2 are related by the n-th approximation relation only if they are indistinguishable in any context for n steps of computation. (They might be distinguishable *after* n steps, but we don't care because the "clock" has run out.) Thus, values are related in the limit only if they are indistinguishable in any context for *any* finite number of steps, *i.e.,* if they are really indistinguishable.

The step-indexed stratification makes it possible to define the semantics of recursive types quite easily. Two values $\texttt{fold}\, v_1$ and $\texttt{fold}\, v_2$ are defined to be related by $\mathcal{V}_n[\![\mu\alpha.\,\tau]\!]\rho$ iff v_1 and v_2 are related by $\mathcal{V}_k[\![[\mu\alpha.\,\tau/\alpha]\tau]\!]\rho$ for all $k < n$. Even though the unfolded type is larger (usually a deal breaker for logical relations, which are typically defined by induction on types), the step index gets smaller, so the definition of the logical relation is well-founded. Moreover, it makes sense for the step index to get smaller, since it takes a step of computation to extract v_i from $\texttt{fold}\, v_i$.

Just as we use steps to stratify logical relations, we can also use them to stratify our quasi-circular possible worlds. We define an "n-level world" inductively to be one whose constituent store relations (the ψ's) are parameterized by $(n-1)$-level worlds. The intuition behind this stratification of worlds is actually very simple: an n-level world describes properties of the current stores that may affect the relatedness of pairs of values for n steps of computation. Since it takes one step of computation just to *inspect* the stores (via a pointer dereference), the relatedness of pairs of values for n steps can only possibly depend on the relatedness of the current stores for $n-1$ steps. Thus, it is fine for an n-level world to be defined as a collection of $(n-1)$-level store relations, *i.e.,* ψ's that only guarantee relatedness of memory contents for $n-1$ steps. And these $(n-1)$-level ψ's, in turn, need only be parameterized by $(n-1)$-level worlds.

4. Step-Indexed Logical Relations for $\mathsf{F}^{\mu!}$

In this section, we present the details of our logical relation for $\mathsf{F}^{\mu!}$ and prove it sound with respect to contextual equivalence.

The basic idea is to give a relational interpretation $\mathcal{V}[\![\tau]\!]$ of a (closed) type τ as a set of tuples of the form (k, W, v_1, v_2), where k is a natural number (called the *step index*), W is a world (as motivated in Section 3), and v_1 and v_2 are values. Informally, $(k, W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]$ says that in any computation running for no more than k steps, v_1 approximates v_2 at the type τ in world W. An important point is that to determine if v_1 approximates v_2 for k steps (at type τ), it suffices for the world W to be a k-level world. That is, the store relations ψ in W need only guarantee relatedness of memory contents for $k-1$ steps, as discussed in Section 3.5. We make the notion of a "k-level world" precise in Section 4.1.

Preliminaries In the rest of the paper, the metavariables i, j, k, m, and n all range over natural numbers. We use the metavariable χ to denote sets of tuples of the form (k, W, e_1, e_2) where k is a step index, W is a world, and e_1 and e_2 are closed terms (*i.e.,* terms that may contain locations, but no free type or term variables). Given a set χ of this form, we write χ^{val} to denote the subset of χ such that e_1 and e_2 are values.

As mentioned in Section 3.3, a *world* W is an ordered list (written $\langle w_1, \ldots, w_n \rangle$) of *islands*. An island w is a pair of some current *knowledge* η and a *law* $\mathcal{L}$. The knowledge η for each island represents the current "state" of the island. It comprises four parts: a *store relation* ψ, which is a set of tuples of the form (k, W, s_1, s_2), where k is a step index, W is a world, and s_1, and s_2 are stores; a *population* V, which is a set of closed values; and two store typings Σ_1 and Σ_2. The domains of Σ_1 and Σ_2 give us the sets of locations that the island "cares about" (a notion we mentioned in Section 3.2). Meanwhile, a law $\mathcal{L}$ is a set of pairs (k, η). If $(k, \eta) \in \mathcal{L}$, it means that, at "time" k (representing the number of steps left on the clock), the knowledge η represents an acceptable state for the island to be in. Below we summarize our notation for ease of reference.

Type Interpretation	χ	::=	$\{(k, W, e_1, e_2), \ldots\}$
Store Relation	ψ	::=	$\{(k, W, s_1, s_2), \ldots\}$
Population	V	::=	$\{v_1, \ldots\}$
Knowledge	η	::=	$(\psi, V, \Sigma_1, \Sigma_2)$
Law	$\mathcal{L}$	::=	$\{(k, \eta), \ldots\}$
Island	w	::=	$(\eta, \mathcal{L})$
World	W	::=	$\langle w_1, \ldots, w_n \rangle$

$$
\begin{array}{rcl}
\mathit{CandAtom}_k & \stackrel{\text{def}}{=} & \{(j, W, e_1, e_2) \mid j < k \wedge W \in \mathit{CandWorld}_j\} \\
\mathit{CandType}_k & \stackrel{\text{def}}{=} & \mathcal{P}(\mathit{CandAtom}_k^{\mathsf{val}}) \\
\mathit{CandStoreAtom}_k & \stackrel{\text{def}}{=} & \{(j, W, s_1, s_2) \mid j < k \wedge W \in \mathit{CandWorld}_j\} \\
\mathit{CandStoreRel}_k & \stackrel{\text{def}}{=} & \mathcal{P}(\mathit{CandStoreAtom}_k) \\
\mathit{CandKnowledge}_k & \stackrel{\text{def}}{=} & \mathit{CandStoreRel}_k \times \mathit{Population} \\
 & & \quad \times \mathit{StoreTy} \times \mathit{StoreTy} \\
\mathit{CandLawAtom}_k & \stackrel{\text{def}}{=} & \{(j, \eta) \mid j \leq k \wedge \eta \in \mathit{CandKnowledge}_j\} \\
\mathit{CandLaw}_k & \stackrel{\text{def}}{=} & \mathcal{P}(\mathit{CandLawAtom}_k) \\
\mathit{CandIsland}_k & \stackrel{\text{def}}{=} & \mathit{CandKnowledge}_k \times \mathit{CandLaw}_k \\
\mathit{CandWorld}_k & \stackrel{\text{def}}{=} & \{W \in (\mathit{CandIsland}_k)^n \mid n \geq 0\}
\end{array}
\qquad
\begin{array}{rcll}
\mathit{CandAtom}_\omega & \stackrel{\text{def}}{=} & \bigcup_{k\geq 0} \mathit{CandAtom}_k & \\
\mathit{CandType}_\omega & \stackrel{\text{def}}{=} & \mathcal{P}(\mathit{CandAtom}_\omega^{\mathsf{val}}) \supseteq \bigcup_{k\geq 0} \mathit{CandType}_k & \\
 & & & \\
\lfloor \chi \rfloor_k & \stackrel{\text{def}}{=} & \{(j, W, e_1, e_2) \mid j < k \wedge (j, W, e_1, e_2) \in \chi\} & \\
\lfloor \psi \rfloor_k & \stackrel{\text{def}}{=} & \{(j, W, s_1, s_2) \mid j < k \wedge (j, W, s_1, s_2) \in \psi\} & \\
\lfloor \eta \rfloor_k & \stackrel{\text{def}}{=} & (\lfloor \psi \rfloor_k, V, \Sigma_1, \Sigma_2) & \text{where } \eta = (\psi, V, \Sigma_1, \Sigma_2) \\
\lfloor \mathcal{L} \rfloor_k & \stackrel{\text{def}}{=} & \{(j, \eta) \mid j \leq k \wedge (j, \eta) \in \mathcal{L}\} & \\
\lfloor w \rfloor_k & \stackrel{\text{def}}{=} & (\lfloor \eta \rfloor_k, \lfloor \mathcal{L} \rfloor_k) & \text{where } w = (\eta, \mathcal{L}) \\
\lfloor W \rfloor_k & \stackrel{\text{def}}{=} & \langle \lfloor w_1 \rfloor_k, \ldots, \lfloor w_n \rfloor_k \rangle & \text{where } W = \langle w_1, \ldots, w_n \rangle
\end{array}
$$

$$
\begin{array}{rcl}
(\psi', V', \Sigma_1', \Sigma_2') \sqsupseteq (\psi, V, \Sigma_1, \Sigma_2) & \stackrel{\text{def}}{=} & V' \supseteq V \wedge \Sigma_1' \supseteq \Sigma_1 \wedge \Sigma_2' \supseteq \Sigma_2 \\
(\eta', \mathcal{L}') \sqsupseteq (\eta, \mathcal{L}) & \stackrel{\text{def}}{=} & \eta' \sqsupseteq \eta \wedge \mathcal{L}' = \mathcal{L} \\
\langle w_1', \ldots, w_{n+m}' \rangle \sqsupseteq \langle w_1, \ldots, w_n \rangle & \stackrel{\text{def}}{=} & m \geq 0 \wedge \forall i \in \{1, \ldots, n\}.\ w_i' \sqsupseteq w_i \\
(j, W') \sqsupseteq (k, W) & \stackrel{\text{def}}{=} & j \leq k \wedge W' \sqsupseteq \lfloor W \rfloor_j \wedge W' \in \mathit{World}_j \wedge W \in \mathit{World}_k \\
(j, W') \sqsupset (k, W) & \stackrel{\text{def}}{=} & j < k \wedge (j, W') \sqsupseteq (k, W)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Atom}[\tau_1, \tau_2]_k & \stackrel{\text{def}}{=} & \{(j, W, e_1, e_2) \in \mathit{CandAtom}_k \mid W \in \mathit{World}_j \wedge \Sigma_1(W) \vdash e_1 : \tau_1 \wedge \Sigma_2(W) \vdash e_2 : \tau_2\} \\
\mathit{Type}[\tau_1, \tau_2]_k & \stackrel{\text{def}}{=} & \{\chi \in \mathcal{P}(\mathit{Atom}[\tau_1, \tau_2]_k^{\mathsf{val}}) \mid \forall (j, W, v_1, v_2) \in \chi.\ \forall (j', W') \sqsupseteq (j, W).\ (j', W', v_1, v_2) \in \chi\} \\
\mathit{StoreAtom}_k & \stackrel{\text{def}}{=} & \{(j, W, s_1, s_2) \in \mathit{CandStoreAtom}_k \mid W \in \mathit{World}_j\} \\
\mathit{StoreRel}_k & \stackrel{\text{def}}{=} & \{\psi \in \mathcal{P}(\mathit{StoreAtom}_k) \mid \forall (j, W, s_1, s_2) \in \psi.\ \forall (i, W') \sqsupseteq (j, W).\ (i, W', s_1, s_2) \in \psi\} \\
\mathit{Knowledge}_k & \stackrel{\text{def}}{=} & \{(\psi, V, \Sigma_1, \Sigma_2) \in \mathit{CandKnowledge}_k \mid \psi \in \mathit{StoreRel}_k \wedge \\
 & & \quad \forall s_1, s_2, s_1', s_2'. \\
 & & \quad\quad (\forall l \in \mathrm{dom}(\Sigma_1).\ s_1(l) = s_1'(l) \ \wedge\ \forall l \in \mathrm{dom}(\Sigma_2).\ s_2(l) = s_2'(l)) \implies \\
 & & \quad\quad \forall j, W.\ (j, W, s_1, s_2) \in \psi \iff (j, W, s_1', s_2') \in \psi\} \\
\mathit{LawAtom}_k & \stackrel{\text{def}}{=} & \{(j, \eta) \in \mathit{CandLawAtom}_k \mid \eta \in \mathit{Knowledge}_j\} \\
\mathit{Law}_k & \stackrel{\text{def}}{=} & \{\mathcal{L} \in \mathcal{P}(\mathit{LawAtom}_k) \mid \forall (j, \eta) \in \mathcal{L}.\ \forall i < j.\ (i, \lfloor \eta \rfloor_i) \in \mathcal{L}\} \\
\mathit{Island}_k & \stackrel{\text{def}}{=} & \{(\eta, \mathcal{L}) \in \mathit{Knowledge}_k \times \mathit{Law}_k \mid (k, \eta) \in \mathcal{L}\} \\
\mathit{World}_k & \stackrel{\text{def}}{=} & \{W \in (\mathit{Island}_k)^n \mid n \geq 0 \wedge \\
 & & \quad \forall a, b \in \{1, \ldots, n\}.\ a \neq b \implies \\
 & & \quad\quad \mathrm{dom}(W[a].\Sigma_1) \# \mathrm{dom}(W[b].\Sigma_1) \wedge \mathrm{dom}(W[a].\Sigma_2) \# \mathrm{dom}(W[b].\Sigma_2)\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Atom}[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \bigcup_{k\geq 0} \mathit{Atom}[\tau_1, \tau_2]_k \\
\mathit{Type}[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \{\chi \in \mathit{CandType}_\omega \mid \forall k \geq 0.\ \lfloor \chi \rfloor_k \in \mathit{Type}[\tau_1, \tau_2]_k\} \quad \supseteq \quad \bigcup_{k\geq 0} \mathit{Type}[\tau_1, \tau_2]_k
\end{array}
$$

Figure 3. Auxiliary Definitions: Candidate Sets, k-Approximation, World Extension, and Well-Formedness Conditions

If $W = \langle w_1, \ldots, w_n \rangle$ and $1 \leq j \leq n$, we write $W[j]$ as shorthand for w_j. If $w = (\eta_i, \mathcal{L}_i)$ where $\eta_i = (\psi_i, V_i, \Sigma_{i1}, \Sigma_{i2})$, we use the following shorthand to extract various elements out of the island w:

$$
\begin{array}{rclcrcl}
w.\eta & \equiv & \eta_i & \qquad & w.V & \equiv & V_i \\
w.\mathcal{L} & \equiv & \mathcal{L}_i & & w.\Sigma_1 & \equiv & \Sigma_{i1} \\
w.\psi & \equiv & \psi_i & & w.\Sigma_2 & \equiv & \Sigma_{i2}
\end{array}
$$

If W is a world with n islands, we also use the following shorthand:

$$
\begin{array}{rcl}
\Sigma_1(W) & \stackrel{\text{def}}{=} & \bigcup_{1\leq j\leq n} W[j].\Sigma_1 \\
\Sigma_2(W) & \stackrel{\text{def}}{=} & \bigcup_{1\leq j\leq n} W[j].\Sigma_2
\end{array}
$$

We write Val for the set of all values, Store for the set of all stores (finite maps from locations to values), and $\mathit{StoreTy}$ for the set of store typings (finite maps from locations to closed types). We write $\mathit{Population}$ for the set of all subsets of Val. Finally, we write $S_1 \# S_2$ to denote that the sets S_1 and S_2 are disjoint.

4.1 Well-Founded, Well-Formed Worlds and Relations

Notice that we cannot naïvely construct a set-theoretic model based on the above intentions since the worlds we wish to construct are (effectively) lists of store relations and store relations are themselves parameterized by worlds (as discussed in Section 3.4). If we ignore islands, laws, populations, and store typings for the moment, and simply model worlds as lists of store relations, we are led to the following specification which captures the essence of the problem:

$$
\begin{array}{rcl}
\mathit{StoreRel} & = & \mathcal{P}(\mathbb{N} \times \mathit{World} \times \mathit{Store} \times \mathit{Store}) \\
\mathit{World} & = & \mathit{StoreRel}^n
\end{array}
$$

A simple diagonalization argument shows that the set $\mathit{StoreRel}$ has an inconsistent cardinality (*i.e.*, it is an ill-founded recursive definition).

We eliminate the inconsistency by stratifying our definition via the step index. To do so, we first construct *candidate* sets, which are well-founded sets of our intended form. We then construct proper notions of worlds, islands, laws, store relations, and so

on, by filtering the candidate sets through some additional well-formedness constraints.

Figure 3 (top left) defines our candidate sets by induction on k. First, note that elements of $CandAtom_k$ and $CandStoreAtom_k$ are tuples with step index j strictly less than k. Hence, our candidate sets are well-defined at all steps. Next, note that elements of $CandLawAtom_k$ are tuples with step index $j \leq k$. Informally, this is because a k-level law should be able to govern the current knowledge (*i.e.*, the knowledge at the present time when we have k steps left to execute), not just the knowledge in the future when we have strictly fewer steps left.

While our candidate sets establish the existence of sets of our intended form, our worlds and type relations will need to be well-behaved in other ways. There are key constraints associated with atoms, types, store relations, knowledge, laws, islands, and worlds that will be enforced in our final definitions. To specify these constraints we need some additional functions and predicates.

For any set χ and any set ψ, we define the k-approximation of the set (written $\lfloor\chi\rfloor_k$ and $\lfloor\psi\rfloor_k$, respectively) as the subset of its elements whose indices are *strictly less* than k (see Figure 3, top right). Meanwhile, for any set $\mathcal{L}$, we define the k-approximation of the set (written $\lfloor\mathcal{L}\rfloor_k$) as the subset of its elements whose indices are *less than or equal* to k. We extend these k-approximation notions to knowledge η, islands w, and worlds W (written $\lfloor\eta\rfloor_k$, $\lfloor w\rfloor_k$, and $\lfloor W\rfloor_k$, respectively) by applying k-approximation to their constituent parts. Note that each of the k-approximation functions yields elements of $CandX_k$ where X denotes the appropriate semantic object.

Next, we define the notion of *world extension* (see Figure 3, middle). We write $(j, W') \sqsupseteq (k, W)$ (where $\sqsupseteq$ is pronounced "extends") if W is a world that is good for k steps (*i.e.*, $W \in World_k$, see below), W' is a good world for $j \leq k$ steps ($W' \in World_j$), and W' extends $\lfloor W\rfloor_j$ (written $W' \sqsupseteq \lfloor W\rfloor_j$). Recall from Section 3.3 that future worlds accessible from W may have new islands added to the end of the list. Furthermore, for each island $w \in \lfloor W\rfloor_j$, the island w' in the same position in W' must extend w. Here we require that $w'.\mathcal{L} = w.\mathcal{L}$ since an island's law cannot be amended in future worlds (see Section 3.3). We also require that $w'.\eta \sqsupseteq w.\eta$, which says that the island's population may grow ($w'.V \supseteq w.V$), as may the sets of locations that the island cares about ($w'.\Sigma_1 \supseteq w.\Sigma_1$ and $w'.\Sigma_2 \supseteq w.\Sigma_2$). Though it may seem from the definition of knowledge extension in Figure 3 that we do not impose any constraints on $w'.\psi$, this is not the case. As explained in Section 3.3, an island's law should govern what the island's future store relations, populations, and locations of concern may look like. The requirement $W' \in World_j$ (which we discuss below) ensures that the future knowledge $w'.\eta$ obeys the law $w'.\mathcal{L}$.

Figure 3 (bottom) defines our various semantic objects, again by induction on k. These definitions serve to filter their corresponding candidate sets. We proceed now to discuss each of these filtering constraints.

Following Pitts [20], our model is built from syntactically well-typed terms. Thus, we define $Atom[\tau_1, \tau_2]_k$ as the set of tuples (j, W, e_1, e_2) where $\Sigma_1(W) \vdash e_1 : \tau_1$ and $\Sigma_2(W) \vdash e_2 : \tau_2$, and $j < k$. (Recall that $\Sigma_i(W)$ denotes the "global" store typing—*i.e.*, the union of the Σ_i components of all the islands in W.) We also require the world W to be a member of $World_j$.

We define $Type[\tau_1, \tau_2]_k$ as those sets $\chi \subseteq Atom[\tau_1, \tau_2]_k^{\mathsf{val}}$ that are closed under world extension. Informally, if v_1 and v_2 are related for k steps in world W, then v_1 and v_2 should also be related for j steps in any future world W' such that (j, W') is accessible from (*i.e.*, extends) (k, W). We define $StoreRel_k$ as the set of all $\psi \subseteq StoreAtom_k \subseteq CandStoreAtom_k$ that are closed under world extension. This property is critical in ensuring that we can extend a world with new islands without fear of breaking the store properties from the old islands.

$Knowledge_k$ is the set of all tuples of the form $(\psi, V, \Sigma_1, \Sigma_2) \in CandKnowledge_k$ such that $\psi \in StoreRel_k$. As mentioned above, the domains of Σ_1 and Σ_2 contain the locations that an island cares about. What this means is that when determining whether two stores s_1 and s_2 belong to the store relation ψ, we cannot depend upon the contents of any location in store s_1 that is not in $\mathrm{dom}(\Sigma_1)$ or on the contents of any location in s_2 that is not in $\mathrm{dom}(\Sigma_2)$. Thus, Σ_1 and Σ_2 essentially serve as *accessibility maps* [7]. While Benton and Leperchey's accessibility maps are functions from stores to subsets of Loc, our accessibility maps are essentially sets of locations that are allowed to grow over time.

We define Law_k as the set of laws $\mathcal{L}$ such that for all $(j, \eta) \in \mathcal{L}$ we have that $\eta \in Knowledge_j$. Furthermore, we require that the sets $\mathcal{L}$ be closed under decreasing step index—that is, if some knowledge η obeys law $\mathcal{L}$ for j steps, then it must be the case that at any future time, when we have $i < j$ steps left, the i-approximation of knowledge η still obeys the law $\mathcal{L}$.

$Island_k$ is the set of all pairs $(\eta, \mathcal{L}) \in (Knowledge_k \times Law_k)$ such that the knowledge η obeys the law $\mathcal{L}$ at the current time denoted by step index k — *i.e.*, $(k, \eta) \in \mathcal{L}$.

Finally, we define $World_k$ as the set of all $W \in (Island_k)^n$. We also require that the sets of locations that each island $W[a]$ cares about are disjoint from the sets of locations that any other island $W[b]$ cares about, thus ensuring separation of islands.

4.2 Relational Interpretations of Types

Figure 4 (top) gives the definition of our logical relations for $\mathsf{F}^{\mu!}$. The relations $\mathcal{V}_n[\![\tau]\!]\,\rho$ are defined by induction on n and nested induction on the type τ. We use the metavariable ρ to denote type substitutions. A type substitution ρ is a finite map from type variables α to triples (χ, τ_1, τ_2) where τ_1 and τ_2 are closed types, and χ is a relational interpretation in $Type[\tau_1, \tau_2]$. If $\rho(\alpha) = (\chi, \tau_1, \tau_2)$, then $\rho_1(\alpha)$ denotes τ_1 and $\rho_2(\alpha)$ denotes τ_2.

Note that, by the definition of $\mathcal{V}_n[\![\tau]\!]\,\rho$, if $(k, W, v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\,\rho$, then $k < n$, $W \in World_k$, and $\Sigma_1(W) \vdash v_1 : \rho_1(\tau)$ and $\Sigma_2(W) \vdash v_2 : \rho_2(\tau)$. Most of the relations $\overline{\mathcal{V}}_n[\![\tau]\!]\,\rho$ are straightforward. For instance, the logical relation at type int says that two integers are logically related for any number of steps k and in any world W as long as they are equal. The relations for the other base types unit and bool are similar. The logical relation at type $\tau \times \tau'$ says that two pairs of values are related for k steps in world W if their first and second components are related (each for k steps in world W) at types τ and τ' respectively.

Since functions are suspended computations, their relatedness is defined based on the relatedness of computations (characterized by the relation $\mathcal{E}_n[\![\tau]\!]\,\rho$, discussed below). Two functions $\lambda x\!:\!\rho_1(\tau).\, e_1$ and $\lambda x\!:\!\rho_2(\tau).\, e_2$ are related for k steps in world W at the type $\tau \to \tau'$ if, in any future world W' where there are $j < k$ steps left to execute and we have arguments v_1 and v_2 that are related at the argument type τ, the computations $[v_1/x]e_1$ and $[v_2/x]e_2$ are also related for j steps in world W' at the result type τ' (*i.e.*, they are in the relation $\mathcal{E}_n[\![\tau']\!]\,\rho$). Intuitively, $j < k$ suffices since beta-reduction consumes a step. Parameterizing over an arbitrary future world W' is necessary here in order to ensure closure of the logical relation under world extension.

Before we can specify when two computations are related, we have to describe what it means for two stores to be related. We write $s_1, s_2 :_k W$, denoting that the stores s_1 and s_2 are related for k steps at the world W (see Figure 4, top), if the stores are well-typed with respect to the store typings $\Sigma_1(W)$ and $\Sigma_2(W)$, respectively, and if the stores are considered acceptable by—*i.e.*, they are in the store relations of—all the islands in W at all future times when $j < k$.

$$s_1, s_2 :_k W \stackrel{\text{def}}{=} \vdash s_1 : \Sigma_1(W) \wedge \vdash s_2 : \Sigma_2(W) \wedge \forall w \in W.\, \forall j < k.\, (j, \lfloor W \rfloor_j, s_1, s_2) \in w.\psi$$

$$\mathcal{V}_n[\![\tau]\!]\,\rho = \overline{\mathcal{V}}_n[\![\tau]\!]\,\rho \cap Atom[\rho_1(\tau), \rho_2(\tau)]_n^{\mathsf{val}}$$

$$\overline{\mathcal{V}}_n[\![\alpha]\!]\,\rho = \chi \quad \text{where } \rho(\alpha) = (\chi, \tau_1, \tau_2)$$

$$\overline{\mathcal{V}}_n[\![\mathsf{unit}]\!]\,\rho = \{(k, W, (), ())\}$$

$$\overline{\mathcal{V}}_n[\![\mathsf{int}]\!]\,\rho = \{(k, W, v, v) \mid v \in \mathbb{Z}\}$$

$$\overline{\mathcal{V}}_n[\![\mathsf{bool}]\!]\,\rho = \{(k, W, v, v) \mid v = \mathtt{true} \vee v = \mathtt{false}\}$$

$$\overline{\mathcal{V}}_n[\![\tau \times \tau']\!]\,\rho = \{(k, W, \langle v_1, v_1'\rangle, \langle v_2, v_2'\rangle) \mid (k, W, v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\,\rho \wedge (k, W, v_1', v_2') \in \mathcal{V}_n[\![\tau']\!]\,\rho\}$$

$$\overline{\mathcal{V}}_n[\![\tau \to \tau']\!]\,\rho = \{(k, W, \lambda x : \rho_1(\tau).\, e_1, \lambda x : \rho_2(\tau).\, e_2) \mid \forall (j, W') \sqsupseteq (k, W).\, \forall v_1, v_2.\ (j, W', v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\,\rho \implies (j, W', [v_1/x]e_1, [v_2/x]e_2) \in \mathcal{E}_n[\![\tau']\!]\,\rho\}$$

$$\overline{\mathcal{V}}_n[\![\forall\alpha.\,\tau]\!]\,\rho = \{(k, W, \Lambda\alpha.\, e_1, \Lambda\alpha.\, e_2) \mid \forall (j, W') \sqsupseteq (k, W).\, \forall \tau_1, \tau_2, \chi \in Type[\tau_1, \tau_2].\ (j, W', [\tau_1/\alpha]e_1, [\tau_2/\alpha]e_2) \in \mathcal{E}_n[\![\tau]\!]\,\rho[\alpha \mapsto (\chi, \tau_1, \tau_2)]\}$$

$$\overline{\mathcal{V}}_n[\![\exists\alpha.\,\tau]\!]\,\rho = \{(k, W, \mathtt{pack}\,\tau_1, v_1\,\mathtt{as}\,\exists\alpha.\,\rho_1(\tau), \mathtt{pack}\,\tau_2, v_2\,\mathtt{as}\,\exists\alpha.\,\rho_2(\tau)) \mid \exists\chi \in Type[\tau_1, \tau_2].\ (k, W, v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\,\rho[\alpha \mapsto (\chi, \tau_1, \tau_2)]\}$$

$$\overline{\mathcal{V}}_n[\![\mu\alpha.\,\tau]\!]\,\rho = \{(k, W, \mathtt{fold}\,v_1, \mathtt{fold}\,v_2) \mid k < n \wedge \forall j < k.\, (j, \lfloor W \rfloor_j, v_1, v_2) \in \mathcal{V}_k[\![[\mu\alpha.\,\tau/\alpha]\tau]\!]\,\rho\}$$

$$\overline{\mathcal{V}}_n[\![\mathsf{ref}\,\tau]\!]\,\rho = \{(k, W, l_1, l_2) \mid k < n \wedge w_{\mathsf{ref}}(k, \rho, \tau, l_1, l_2) \in W\}$$

$$w_{\mathsf{ref}}(k, \rho, \tau, l_1, l_2) = (\eta, \mathcal{L}) \quad \text{where } \eta = (\psi, \{\}, \{l_1 : \rho_1(\tau)\}, \{l_2 : \rho_2(\tau)\}),\ \psi = \{(j, W', s_1, s_2) \mid (j, W', s_1(l_1), s_2(l_2)) \in \mathcal{V}_k[\![\tau]\!]\,\rho\},\ \mathcal{L} = \{(j, \lfloor\eta\rfloor_j) \mid j \le k\}$$

$$\mathcal{E}_n[\![\tau]\!]\,\rho = \{(k, W, e_1, e_2) \in Atom[\rho_1(\tau), \rho_2(\tau)]_n \mid \forall j < k.\, \forall s_1, s_2, s_1', v_1.\ s_1, e_1 \longmapsto^j s_1', v_1 \wedge s_1, s_2 :_k W \implies \exists s_2', v_2, W'.\ (k - j, W') \sqsupseteq (k, W) \wedge s_2, e_2 \longmapsto^* s_2', v_2 \wedge s_1', s_2' :_{k-j} W' \wedge (k - j, W', v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\,\rho\}$$

$$\mathcal{V}[\![\tau]\!]\,\rho = \bigcup_{n \ge 0} \mathcal{V}_n[\![\tau]\!]\,\rho \qquad \mathcal{E}[\![\tau]\!]\,\rho = \bigcup_{n \ge 0} \mathcal{E}_n[\![\tau]\!]\,\rho$$

$$\mathcal{D}[\![\cdot]\!] = \{\emptyset\}$$
$$\mathcal{D}[\![\Delta, \alpha]\!] = \{\rho[\alpha \mapsto (\chi, \tau_1, \tau_2)] \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge \chi \in Type[\tau_1, \tau_2]\}$$

$$\mathcal{G}[\![\cdot]\!]\,\rho = \{(k, W, \emptyset) \mid W \in World_k\}$$
$$\mathcal{G}[\![\Gamma, x : \tau]\!]\,\rho = \{(k, W, \gamma[x \mapsto (v_1, v_2)]) \mid (k, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\,\rho \wedge (k, W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\,\rho\}$$

$$\mathcal{S}[\![\Sigma]\!] = \{(k, W) \mid \forall (l : \tau) \in \Sigma.\, (k, W, l, l) \in \mathcal{V}[\![\mathsf{ref}\,\tau]\!]\,\emptyset\}$$

$$\Delta; \Gamma; \Sigma \vdash e_1 \preceq^{log} e_2 : \tau \stackrel{\text{def}}{=} \Delta; \Gamma; \Sigma \vdash e_1 : \tau \wedge \Delta; \Gamma; \Sigma \vdash e_2 : \tau \wedge \forall k \ge 0.\, \forall \rho, \gamma, W.\ \rho \in \mathcal{D}[\![\Delta]\!] \wedge (k, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\,\rho \wedge (k, W) \in \mathcal{S}[\![\Sigma]\!] \implies (k, W, \rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{E}[\![\tau]\!]\,\rho$$

$$\Delta; \Gamma; \Sigma \vdash e_1 \approx^{log} e_2 : \tau \stackrel{\text{def}}{=} \Delta; \Gamma; \Sigma \vdash e_1 \preceq^{log} e_2 : \tau \wedge \Delta; \Gamma; \Sigma \vdash e_2 \preceq^{log} e_1 : \tau$$

Figure 4. Step-Indexed Logical Relations for $\mathsf{F}^{\mu!}$

The relation $\mathcal{E}_n[\![\tau]\!]\,\rho$ specifies when two computations are logically related. Two closed, well-typed terms e_1 and e_2 are related for k steps at the type τ in world W if, given two initial stores s_1 and s_2 that are related for k steps at world W, if s_1, e_1 evaluates to s_1', v_1 in $j < k$ steps then the following conditions hold. First, s_2, e_2 must evaluate to some s_2', v_2 in *any* number of steps. (For details on why the number of steps e_2 takes is irrelevant, see Ahmed [1].) Second, there must exist a world $W' \in World_{k-j}$ that extends the world W. Third, the final stores s_1' and s_2' must be related for the remaining $k - j$ steps at world W'. Fourth, the values v_1 and v_2 must be related for $k - j$ steps in the world W' at the type τ. Notice the asymmetric nature of the relation on computations: if s_1, e_1 terminates, then s_2, e_2 must also terminate. Hence, our relations $\mathcal{V}_n[\![\tau]\!]\,\rho$ model logical approximation rather than logical equivalence.

The cases of the logical relation for $\forall\alpha.\tau$ and $\exists\alpha.\tau$ are essentially standard. The former involves *parameterizing* over an arbitrary relational interpretation χ of α, and the latter involves *choosing* an arbitrary relational interpretation χ of α. The way the worlds are manipulated follows in the style of the other rules. The logical relation for $\mu\alpha.\tau$ is very similar to previous step-indexed accounts of recursive types, as described in Section 3.5. (Note that, although the type gets larger on the r.h.s. of the definition, the step index gets smaller, so the definition is well-founded.)

Any two locations related at a type $\mathsf{ref}\,\tau$ are publicly accessible references. For reasoning about such *visible* locations, existing logical relations methods usually employ some mechanism that is distinct from the machinery used to reason about local or *hidden* state. Since there always exists a bijection between the visible locations of the two computations, the mechanism usually involves having a special portion of the world that tracks the bijection between visible locations as well as the type τ of their contents. Unlike previous methods, our worlds have no specialized machinery for reasoning about visible locations. Our technique for modeling (publicly accessible) references is simply a mode of use of our mechanism for reasoning about local state.

Intuitively, two locations l_1 and l_2 should be related at the type $\mathsf{ref}\,\tau$ in world W for k steps if, given any two stores s_1 and s_2 that are related for k steps at world W, the contents of these locations, *i.e.,* $s_1(l_1)$ and $s_2(l_2)$, are related for $k - 1$ steps at the type τ. To enforce this requirement, we simply install a special island w_{ref} that only cares about the one location l_1 in s_1 and the one location l_2 in s_2. Furthermore, w_{ref} has an empty population and a law that says the population should remain empty in future worlds. Finally, the island's *fixed* store relation ψ relates all stores s_1 and s_2 whose contents at locations l_1 and l_2, respectively, are related at type τ for $j < k$ steps. Here $j < k$ suffices since pointer dereferencing consumes a step (see Section 3.5).

The definitions of logical approximation and equivalence for open terms are given at the bottom of Figure 4. These definitions rely on the relational semantics ascribed to the contexts Δ, Γ, Σ, which we discuss next.

We say a type substitution ρ belongs to the relational interpretation of Δ if $\mathrm{dom}(\rho) = \Delta$, and whenever $\rho(\alpha) = (\chi, \tau_1, \tau_2)$, χ is a well-formed relational interpretation (*i.e.,* $\chi \in Type[\tau_1, \tau_2]$).

We let the metavariable γ range of relational value substitutions. These are finite maps from term variables x to pairs of values (v_1, v_2). If $\gamma(x) = (v_1, v_2)$, then $\gamma_1(x)$ denotes v_1 and $\gamma_2(x)$ denotes v_2. We say γ belongs to the relational interpretation of Γ for k steps at world W (written $(k, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\,\rho$, where $FTV(\Gamma) \subseteq \mathrm{dom}(\rho)$), if $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$, and the values $\gamma_1(x)$ and $\gamma_2(x)$ are related for k steps in world W at type $\Gamma(x)$.

We say a world W satisfies a store typing Σ for k steps (written $(k, W) \in \mathcal{S}[\![\Sigma]\!]$) if W contains an island of the form $w_{\mathsf{ref}}(k, \emptyset, \tau, l, l)$ for each $(l : \tau) \in \Sigma$—*i.e.,* if l is related to itself for k steps in world W at type $\mathsf{ref}\,\tau$.

We write $\Delta;\Gamma;\Sigma \vdash e_1 \preceq^{log} e_2 : \tau$ (pronounced "e_1 logically approximates e_2") to mean that for all k, given a type substitution $\rho \in \mathcal{D}[\![\Delta]\!]$ and a relational value substitution γ such that $(k, W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho$, where the world W satisfies Σ for k steps, the closed terms $\rho_1(\gamma_1(e_1))$ and $\rho_2(\gamma_2(e_2))$ are related for k steps in world W at the type τ. Finally, we say e_1 and e_2 are logically equivalent, written $\Delta;\Gamma;\Sigma \vdash e_1 \approx^{log} e_2 : \tau$, if they logically approximate each other.

4.3 Fundamental Property & Soundness of Logical Relation

Here we state some of the main properties of our logical relation and sketch interesting aspects of the proofs. Further details of the meta-theory are given in the online technical appendix [3].

Lemma 4.1 (Closure Under World Extension)

Let $\Delta \vdash \tau$ and $\rho \in \mathcal{D}[\![\Delta]\!]$. If $(k, W, v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\rho$ and $(j, W') \sqsupseteq (k, W)$, then $(j, W', v_1, v_2) \in \mathcal{V}_n[\![\tau]\!]\rho$.

Proof: By induction on n and nested induction on $\Delta \vdash \tau$. □

An important property of logical approximation is that it is a *precongruence*, *i.e.,* it is *compatible* with all the constructs of the language (see *e.g.,* Pitts [20]). We state these compatibility lemmas, and give detailed proofs of the ones involving references, in the online technical appendix [3]. The most involved cases are those for allocation (`ref`) and assignment, which we discuss below. Proofs of compatibility lemmas that do not involve references essentially follow the proofs given in Ahmed [1]—although we must now deal with additional hypotheses and goals involving stores and worlds, this does not complicate the proofs in any fundamental way.

The compatibility property for `ref` says that if $\Delta;\Gamma;\Sigma \vdash e_1 \preceq^{log} e_2 : \tau$ then $\Delta;\Gamma;\Sigma \vdash \texttt{ref}\, e_1 \preceq^{log} \texttt{ref}\, e_2 : \mathsf{ref}\,\tau$. In the proof, we find ourselves at a point where we have stores $s_1, s_2 :_k W$ and we allocate locations $l_1 \notin \mathrm{dom}(s_1)$ and $l_2 \notin \mathrm{dom}(s_2)$ to hold the values v_1 and v_2 respectively (where we know that $(k, W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho$). To proceed, we define a new world $W' \in World_{k-1}$, which is just $\lfloor W \rfloor_{k-1}$ extended with a new island $w_{\mathsf{ref}}(k-1, \rho, \tau, l_1, l_2)$. In addition to showing that W' is a valid world, which is straightforward, we must also show that (1) $(k-1, W') \sqsupseteq (k, W)$ and (2) $s_1[l_1 \mapsto v_1], s_2[l_2 \mapsto v_2] :_{k-1} W'$.

For (1) we need to show that l_1 and l_2 are distinct from locations that any island $w \in W$ "cares about"—that is, $l_1 \notin \mathrm{dom}(\Sigma_1(W))$ and $l_2 \notin \mathrm{dom}(\Sigma_2(W))$, which follows easily since l_1 and l_2 are fresh for s_1 and s_2. For (2) we must show that for all $w' \in W'$, and $j < k-1$, $(j, \lfloor W' \rfloor_j, s_1[l_1 \mapsto v_1], s_2[l_2 \mapsto v_2]) \in w'.\psi$. If w' is the new island $w_{\mathsf{ref}}(k-1, \rho, \tau, l_1, l_2)$, then the desired result follows from the knowledge that v_1 and v_2 are logically related. If w' is any other island, it must be the $(k-1)$-th approximation of some island $w \in W$. In this case, the desired result follows from closure of $w.\psi$ under world extension, together with the fact that s_i and $s_i[l_i \mapsto v_i]$ are identical when restricted to the domain $w.\Sigma_i$.

The proof of the compatibility lemma for assignment is quite similar to that for `ref`, except that we do not add a new island to W since we know that W already contains an island $w_{\mathsf{ref}}(k, \rho, \tau, l_1, l_2)$ where l_1 and l_2 are the locations being updated.

Theorem 4.2 (Fundamental Property)

If $\Delta;\Gamma;\Sigma \vdash e : \tau$ then $\Delta;\Gamma;\Sigma \vdash e \preceq^{log} e : \tau$.

Proof: By induction on the derivation of $\Delta;\Gamma;\Sigma \vdash e : \tau$. Each case follows from the corresponding compatibility lemma. □

Soundness To show that the logical relation is sound with respect to contextual approximation, we need an additional property we call *store parametricity*. This property says that if $\vdash s : \Sigma$ and $W \in World_k$ is a world comprising one w_{ref} island for each location in Σ—*i.e.,* if $\Sigma = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ and $W = \langle w_1, \ldots, w_n \rangle$, where each $w_i = w_{\mathsf{ref}}(k, \emptyset, \tau_i, l_i, l_i)$)—then $s, s :_k W$.

Notice that, to prove store parametricity, we need to show that for each $(l_i : \tau_i) \in \Sigma$, the value stored at location l_i in store s is related to itself at the type τ_i (*i.e.,* $(k, W, s(l_i), s(l_i)) \in \mathcal{V}[\![\tau_i]\!]\emptyset$). Unfortunately, the latter does not follow from the Fundamental Property, which only allows us to conclude from $\cdot;\cdot;\Sigma \vdash s(l_i) : \tau_i$ that $(k, W, s(l_i), s(l_i)) \in \mathcal{E}[\![\tau_i]\!]\emptyset$.

What we need is the notion of *logical value approximation*, $\Delta;\Gamma;\Sigma \vdash v_1 \preceq^{log}_{\mathsf{val}} v_2 : \tau$, which we define exactly as $\Delta;\Gamma;\Sigma \vdash v_1 \preceq^{log} v_2 : \tau$ except that the $\mathcal{E}[\![\tau]\!]\rho$ at the end of that definition is replaced with $\mathcal{V}[\![\tau]\!]\rho$. Now we can prove that any well-typed value is related to itself in the appropriate value relation $\mathcal{V}[\![\tau]\!]\rho$, not just in the computation relation $\mathcal{E}[\![\tau]\!]\rho$ as established by the Fundamental Property. Specifically, we show that $\Delta;\Gamma;\Sigma \vdash v : \tau$ implies $\Delta;\Gamma;\Sigma \vdash v \preceq^{log}_{\mathsf{val}} v : \tau$. (The proof is by induction on $\Delta;\Gamma;\Sigma \vdash v : \tau$ and for each case the proof is similar to that of the corresponding compatibility lemma.) With this lemma in hand, store parametricity follows easily.

Theorem 4.3 (Soundness w.r.t. Contextual Approximation)

If $\Delta;\Gamma;\Sigma \vdash e_1 \preceq^{log} e_2 : \tau$ then $\Delta;\Gamma;\Sigma \vdash e_1 \preceq^{ctx} e_2 : \tau$.

Proof: Suppose $\vdash C : (\Delta;\Gamma;\Sigma \vdash \tau) \Rightarrow (\cdot;\cdot;\Sigma' \vdash \tau')$, $\vdash s : \Sigma'$, and $s, C[e_1] \longmapsto^k s_1, v_1$. We must show that $s, C[e_2] \Downarrow$.

If $\Sigma = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, let $W = \langle w_1, \ldots, w_n \rangle$, where each $w_i = w_{\mathsf{ref}}(k+1, \emptyset, \tau_i, l_i, l_i)$. By the compatibility lemmas, we can show $\cdot;\cdot;\Sigma' \vdash C[e_1] \preceq^{log} C[e_2] : \tau'$. Hence, noting that $(k+1, W) \in \mathcal{S}[\![\Sigma']\!]$, we have $(k+1, W, C[e_1], C[e_2]) \in \mathcal{E}[\![\tau']\!]\emptyset$. Since $s, s :_{k+1} W$ (by store parametricity) and $s, C[e_1] \longmapsto^k s_1, v_1$ (from the premise), it follows that $s, C[e_2] \Downarrow$. □

5. Examples

In this section we present a number of examples demonstrating applications of our method. Our examples do not make use of recursive types (or even recursion), but Ahmed's prior work, which we build on, gives several examples that do [1]. We will walk through the proof for the first example in detail. For the remaining ones, we only sketch the central ideas, mainly by giving suitable island definitions and type interpretations. Full proofs for these examples and others appear in the online technical appendix [3].

5.1 Name Generator

Our first example is perhaps the simplest possible state-dependent ADT, a generator for fresh names. Nevertheless, it captures the essence of the `Symbol` example from the introduction:

$$e = \texttt{let}\, x = \texttt{ref}\, 0\, \texttt{in}\\ \quad \texttt{pack}\,\mathsf{int}, \langle \lambda z : \mathsf{unit}.\, (++x), \lambda z : \mathsf{int}.\, (z \leq\, !x) \rangle\, \texttt{as}\, \sigma$$

where $\sigma = \exists\alpha.\, (\mathsf{unit} \to \alpha) \times (\alpha \to \mathsf{bool})$ and $(++x)$ abbreviates the expression $(x := !x+1; !x)$, and `let` is encoded in the standard way (using function application). The package defines an abstract type α of names and provides two operations: the first one returns a fresh name on each invocation, and the second one checks that any value of type α it is given is a "valid" name, *i.e.,* one that was previously generated by the first operation.

Names are represented as integers, and the local counter x stores the highest value that has been used so far. The intended invariant of this implementation is that no value of type α ever has a representation that is greater than the current content of x. Under this invariant, we should be able to prove that the second operation, which dynamically checks this property, always returns `true`.

To prove this, we show that e is equivalent to a second expression e', identical to e, except that the dynamic check $(z \leq\, !x)$ is eliminated and replaced by `true`. We only show the one direction, $\vdash e \preceq^{log} e' : \sigma$. The other direction is proven analogously.

Because the terms are closed, this only requires showing that $(k_0, W_0, e, e') \in \mathcal{E}[\![\sigma]\!]\emptyset$ for all $k_0 \geq 0$ and worlds W_0. Assume stores s_0, s_0' $:_{k_0}$ W_0 and the existence of a reduction sequence $s_0, e \longmapsto^{k_1} s_1, v_1$ with $k_1 < k_0$. According to the definition of $\mathcal{E}[\![\sigma]\!]\emptyset$, we need to come up with a reduction $s_0', e' \longmapsto^* s_1', v_1'$ and a world W_1 such that $(k_0 - k_1, W_1) \sqsupseteq (k_0, W_0)$ and:

$$s_1, s_1' :_{k_0-k_1} W_1 \ \wedge\ (k_0 - k_1, W_1, v_1, v_1') \in \mathcal{V}[\![\sigma]\!]\emptyset$$

By inspecting the definition of reduction, we see that

$$s_1 = s_0[l \mapsto 0], \quad v_1 = \mathtt{pack}\,\mathsf{int}, \langle \lambda z.(++l), \lambda z.(z \leq\ !l)\rangle\ \mathtt{as}\,\sigma$$

for some $l \notin \mathrm{dom}(s_0)$. In the same manner, s_0', e' obviously can choose some $l' \notin \mathrm{dom}(s_0')$ and reduce to:

$$s_1' = s_0'[l' \mapsto 0], \quad v_1' = \mathtt{pack}\,\mathsf{int}, \langle \lambda z.(++l'), \lambda z.\mathtt{true}\rangle\ \mathtt{as}\,\sigma$$

We now need to define a suitable island that enables us to show that v_1 and v_1' are related. We know W_0 has the form $\langle w_1, \ldots, w_p\rangle$ for some p. Let W_1 be $\lfloor W_0 \rfloor_{k_0-k_1}$, extended with a new island, w_{p+1}, defined as follows:

$$\begin{array}{rcl} w_{p+1} & = & (\eta^0_{k_0-k_1}, \mathcal{L}_{k_0-k_1}) \\ \eta^n_k & = & (\psi^n_k, V_n, \{l : \mathsf{int}\}, \{l' : \mathsf{int}\}) \\ \psi^n_k & = & \{(j, W, s, s') \in \mathit{StoreAtom}_k \mid s(l) = s'(l') = n\} \\ V_n & = & \{i \mid 1 \leq i \leq n\} \\ \mathcal{L}_k & = & \{(j, \eta^n_j) \in \mathit{LawAtom}_k \mid n \in \mathbb{N}\} \end{array}$$

The population V_n consists of all integers that are "valid" names in the current world, *i.e.,* not greater than the current value of x. We have to show $(k_0 - k_1, W_1) \sqsupseteq (k_0, W_0)$ and $s_1, s_1' :_{k_0-k_1} W_1$. Both are straightforward.

By definition of $\mathcal{V}[\![\exists\alpha.\tau]\!]$, we need to continue by giving a relation $\chi_\alpha \in \mathit{Type}[\mathsf{int}, \mathsf{int}]$, such that:

$$(k_0 - k_1, W_1, \langle \lambda z.(++l), \lambda z.(z \leq\ !l)\rangle, \langle \lambda z.(++l'), \lambda z.\mathtt{true}\rangle) \in \mathcal{V}[\![(\mathsf{unit} \to \alpha) \times (\alpha \to \mathsf{bool})]\!]\,\rho$$

with $\rho = [\alpha \mapsto (\chi_\alpha, \mathsf{int}, \mathsf{int})]$. We choose the following one:

$$\chi_\alpha = \{(j, W, i, i) \in \mathit{Atom}[\mathsf{int}, \mathsf{int}] \mid i \in W[p+1].V\}$$

This interpretation of α depends on the (valid) assumption that it will only be considered at W's that are future worlds of W_1 (in particular, it assumes that the $(p+1)$-th island in W, written $W[p+1]$, is a future version of the w_{p+1} we defined above). We could build this assumption explicitly into the definition of χ_α, but as we will see it is simply not necessary to do so. By virtue of this assumption, a value i is only a valid inhabitant of type α in worlds whose $(p+1)$-th island population contains i, that is, where $!l \geq i$. Note that the relation is closed under world extension because V may only grow over time, as explained in Section 3.3.

By definition of $\mathcal{V}[\![\tau \times \tau']\!]$, it remains to be shown that:

1. $(k_0 - k_1, W_1, \lambda z.(++l), \lambda z.(++l')) \in \mathcal{V}[\![\mathsf{unit} \to \alpha]\!]\,\rho$
2. $(k_0 - k_1, W_1, \lambda z.(z \leq\ !l), \lambda z.\mathtt{true}) \in \mathcal{V}[\![\alpha \to \mathsf{bool}]\!]\,\rho$

For each of these, we assume we begin in some strictly future world W_2 in which $(k_2, W_2) \sqsupset (k_0 - k_1, W_1)$ and $s_2, s_2' :_{k_2} W_2$.

First consider (1). We are given $s_2, (++l) \longmapsto^{k_3} s_3, v_3$ for some $k_3 < k_2$, and it remains to show that $s_2', (++l') \longmapsto^* s_3', v_3'$, such that s_3, s_3' and v_3, v_3' are related in some future world W_3 such that $(k_2 - k_3, W_3) \sqsupseteq (k_2, W_2)$.

From $(k_2, W_2) \sqsupseteq (k_0 - k_1, W_1)$ we know that $W_2[p+1].\mathcal{L} = \lfloor W_1[p+1].\mathcal{L} \rfloor_{k_2} = \mathcal{L}_{k_2}$. From that $(k_2, W_2[p+1].\eta) \in \mathcal{L}_{k_2}$ follows, and hence there exists n such that $W_2[p+1].\eta = \eta^n_{k_2}$. That is, $W_2[p+1].\psi = \psi^n_{k_2}$ and $W_2[p+1].V = V_n$. From $s_2, s_2' :_{k_2} W_2$ and $k_3 < k_2$ we can conclude $(k_3, \lfloor W_2 \rfloor_{k_3}, s_2, s_2') \in \psi^n_{k_2}$ and thus $s_2(l) = s_2'(l') = n$. Consequently, $v_3 = v_3' = n + 1$, $s_3 = s_2[l \mapsto n+1]$, and $s_3' = s_2'[l' \mapsto n+1]$.

Now we choose W_3 to be $\lfloor W_2 \rfloor_{k_2-k_3}$ with its $(p+1)$-th island updated to $(\eta^{n+1}_{k_2-k_3}, \mathcal{L}_{k_2-k_3})$. Again, we have to check the relevant properties, $(k_2 - k_3, W_3) \sqsupseteq (k_2, W_2)$ and $s_3, s_3' :_{k_2-k_3} W_3$, which are straightforward. Last, we have to show that the results v_3, v_3' are related in $\mathcal{V}[\![\alpha]\!]\,\rho$ under this world, *i.e.,* $(k_2 - k_3, W_3, n+1, n+1) \in \chi_\alpha$. Since $n + 1 \in V_{n+1} = W_3[p+1].V$, this is immediate from the definition of χ_α.

Now consider (2). The proof is similar to that for part (1), but simpler. We are given that $(k_2, W_2, v_2, v_2') \in \mathcal{V}[\![\alpha]\!]\,\rho = \chi_\alpha$, and $s_2, (v_2 \leq\ !l) \longmapsto^{k_3} s_3, v_3$ for some $k_3 < k_2$. The main thing to show is that $v_3 = \mathtt{true}$ (we can pick the end world W_3 to just be $\lfloor W_2 \rfloor_{k_2-k_3}$). As in part (1), we can reason that $W_2[p+1].\eta = \eta^n_{k_2}$ for some n, and therefore that $s_2(l) = n$ and, by definition of χ_α, also that $v_2 \leq n$. Hence, $v_2 \leq s_2(l)$, and the desired result follows easily.

5.2 Using `ref` As a Name Generator

An alternative way to implement a name generator is to represent names by locations and rely on generativity of the `ref` operator.

$$\begin{array}{l} e = \mathtt{pack}\,\mathsf{ref\ unit}, \langle \lambda z : \mathsf{unit}.\ (\mathtt{ref}\ ()), \\ \qquad\qquad \lambda p : (\mathsf{ref\ unit} \times \mathsf{ref\ unit}).\ (\mathtt{fst}\,p == \mathtt{snd}\,p)\rangle\ \mathtt{as}\,\sigma \end{array}$$

where $\sigma = \exists\alpha.\,(\mathsf{unit} \to \alpha) \times (\alpha \times \alpha \to \mathsf{bool})$. Here, the second function implements a proper equality operator on names. We want to prove this implementation contextually equivalent to one using integers, as in the previous example:

$$\begin{array}{l} e' = \mathtt{let}\,x = \mathtt{ref}\,0\,\mathtt{in} \\ \quad \mathtt{pack}\,\mathsf{int}, \langle \lambda z : \mathsf{unit}.\ (++x), \\ \qquad\qquad \lambda p : (\mathsf{int} \times \mathsf{int}).\ (\mathtt{fst}\,p = \mathtt{snd}\,p)\rangle\ \mathtt{as}\,\sigma \end{array}$$

Here are a suitable island definition and type interpretation for α:

$$\begin{array}{rcl} w_{p+1} & = & (\eta^{\langle\rangle}_{k_0}, \mathcal{L}_{k_0}) \\ \eta^{\langle l_1,\ldots,l_n\rangle}_k & = & (\psi^n_k, V_{\langle l_1,\ldots,l_n\rangle}, \{l_i : \mathsf{unit} \mid 1 \leq i \leq n\}, \{l' : \mathsf{int}\}) \\ \psi^n_k & = & \{(j, W, s, s') \in \mathit{StoreAtom}_k \mid s'(l') = n\} \\ V_{\langle l_1,\ldots,l_n\rangle} & = & \{\langle l_i, i\rangle \mid 1 \leq i \leq n\} \\ \mathcal{L}_k & = & \{(j, \eta^{\langle l_1,\ldots,l_n\rangle}_j) \in \mathit{LawAtom}_k \mid n \in \mathbb{N}\} \\ \chi_\alpha & = & \{(j, W, l, i) \in \mathit{Atom}[\mathsf{ref\ unit}, \mathsf{int}] \mid \\ & & \qquad \langle l, i\rangle \in W[p+1].V\} \end{array}$$

Here, and in the examples that follow, k_0 represents the current step level, and p the number of islands in the current world W_0, at the point in the proof where we extend W_0 with the island w_{p+1} governing the example's local state. In this example, we assume that all labels in a list $\langle l_1, \ldots, l_n\rangle$ are pairwise disjoint, and l' is a distinguished label, namely the one that has been allocated for x (as in the previous example).

In the definitions above, the population not only records the valid names for e' (as in Section 5.1), but also relates them to the locations allocated by e. The latter are not guessable ahead of time, due to nondeterminism of memory allocation, but the law $\mathcal{L}_k$ is flexible enough to permit any partial bijection between $\{1, \ldots, n\}$ and Loc to evolve over time. We (ab)use term-level pairs $\langle l, i\rangle$ to encode this partial bijection in V. This is sufficient to deduce $i = j$ iff $l_i = l_j$ when proving equivalence of the equality operators.

5.3 Twin Abstraction

Another interesting variation on the generator theme involves the definition of *two* abstract types (we write $\mathtt{pack}\,\tau_1, \tau_2, e\ \mathtt{as}\,\exists\alpha, \beta.\tau$ to abbreviate two nested existentials in the obvious way):

$$\begin{array}{l} e = \mathtt{let}\,x = \mathtt{ref}\,0\,\mathtt{in} \\ \quad \mathtt{pack}\,\mathsf{int}, \mathsf{int}, \langle \lambda z : \mathsf{unit}.\ (++x), \\ \qquad\qquad \lambda z : \mathsf{unit}.\ (++x), \\ \qquad\qquad \lambda p : (\mathsf{int} \times \mathsf{int}).\ (\mathtt{fst}\,p = \mathtt{snd}\,p)\rangle\ \mathtt{as}\,\sigma \end{array}$$

where $\sigma = \exists\alpha,\beta.\,(\mathsf{unit}\to\alpha)\times(\mathsf{unit}\to\beta)\times(\alpha\times\beta\to\mathsf{bool})$. Here we use a single counter to generate names of two types, α and β, and a comparison operator that takes as input names of *different* type. Because both types share the same counter, it appears impossible for a name to belong to both types (either it was generated as a name of type α or of type β but not of both). The example is interesting, however, in that we have no way of knowing the interpretations of α and β ahead of time, since calls to the name generation functions can happen in arbitrary combinations. We can verify our intuition by proving that e is equivalent to an e' where the comparison operator is replaced by $\lambda p\colon(\mathsf{int}\times\mathsf{int}).\,\mathtt{false}$.

The following w and χ definitions enable such a proof:

$$\begin{aligned}
w_{p+1} &= (\eta_{k_0}^{0,\emptyset}, \mathcal{L}_{k_0})\\
\eta_k^{n,S} &= (\psi_k^n, V_{n,S}, \{l:\mathsf{int}\}, \{l':\mathsf{int}\})\\
\psi_k^n &= \{(j,W,s,s') \in \mathit{StoreAtom}_k \mid s(l) = s'(l') = n\}\\
V_{n,S} &= \{\langle 1,i\rangle \mid i\in S\} \cup \{\langle 2,i\rangle \mid i\in\{1,\ldots,n\}\setminus S\}\\
\mathcal{L}_k &= \{(j,\eta_j^{n,S}) \in \mathit{LawAtom}_k \mid n\in\mathbb{N}\wedge S\subseteq\{1,\ldots,n\}\}\\
\chi_\alpha &= \{(j,W,i,i) \in \mathit{Atom}[\mathsf{int},\mathsf{int}] \mid \langle 1,i\rangle \in W[p+1].V\}\\
\chi_\beta &= \{(j,W,i,i) \in \mathit{Atom}[\mathsf{int},\mathsf{int}] \mid \langle 2,i\rangle \in W[p+1].V\}
\end{aligned}$$

The population here is partitioned into the valid names for α and the valid names for β, basically recording the history of calls to the two generator functions. To encode such a disjoint union in V, each value is wrapped in a pair with the first component marking the type it belongs to (1 for α, 2 for β). When proving equivalence of the two comparison operators, the definitions of χ_α, χ_β and $W[p+1].V$ directly imply that the arguments must be from disjoint sets.

5.4 Cell Class

The next example is a more richly-typed variation of the higher-order cell object example of Koutavas and Wand [13]:

$$\begin{aligned}
e = \Lambda\alpha.\,\mathsf{pack}\,\mathsf{ref}\,\alpha, \langle &\lambda x:\alpha.\,\mathtt{ref}\,x,\\
&\lambda r:\mathsf{ref}\,\alpha.\,\mathtt{!}r,\\
&\lambda\langle r,x\rangle:\mathsf{ref}\,\alpha\times\alpha.\,(r\mathtt{:=}x)\rangle\,\mathsf{as}\,\sigma
\end{aligned}$$

where $\sigma = \exists\beta.\,(\alpha\to\beta)\times(\beta\to\alpha)\times(\beta\times\alpha\to\mathsf{unit})$. We use pattern matching notation here merely for clarity and brevity (imagine replacing occurrences of r and x in the third function with `fst` and `snd` projections, respectively, of the argument).

This example generalizes Koutavas and Wand's original version in two ways. First, we parameterize over the cell content type α, which can of course be instantiated with an arbitrary higher type, thus exercising our ability to handle higher-order stored values. Second, instead of just implementing a single object, our example actually models a *class*, where β represents the abstract class type, and the first function acts as a constructor for creating new cell objects. (A subsequent paper by Koutavas and Wand also considers a class-based version of their original example [14], but it is modeled with a Java-like nominal type system, not with existential types.)

Similar to [13], we want to prove this canonical cell implementation equivalent to one using two alternating slots:

$$\begin{aligned}
e' = \Lambda\alpha.\,&\mathsf{pack}\,(\mathsf{ref}\,\mathsf{int}\times(\mathsf{ref}\,\alpha\times\mathsf{ref}\,\alpha)),\\
&\langle\lambda x:\alpha.\,\langle\mathtt{ref}\,1, \langle\mathtt{ref}\,x, \mathtt{ref}\,x\rangle\rangle,\\
&\ \lambda\langle r_0,\langle r_1,r_2\rangle\rangle:(\mathsf{ref}\,\mathsf{int}\times(\mathsf{ref}\,\alpha\times\mathsf{ref}\,\alpha)).\\
&\quad\mathtt{if}\ \mathtt{!}r_0 = 1\ \mathtt{then}\ \mathtt{!}r_1\ \mathtt{else}\ \mathtt{!}r_2,\\
&\ \lambda\langle\langle r_0,\langle r_1,r_2\rangle\rangle,x\rangle:(\mathsf{ref}\,\mathsf{int}\times(\mathsf{ref}\,\alpha\times\mathsf{ref}\,\alpha))\times\alpha.\\
&\quad\mathtt{if}\ \mathtt{!}r_0 = 1\ \mathtt{then}\,(r_0\mathtt{:=}2; r_2\mathtt{:=}x)\\
&\qquad\mathtt{else}\,(r_0\mathtt{:=}1; r_1\mathtt{:=}x)\rangle\,\mathsf{as}\,\sigma
\end{aligned}$$

When e or e' is instantiated with a type argument, neither one immediately allocates any new state. Correspondingly, no island is introduced at that point in the proof. Rather, a new island is added to the world at each call to the classes' constructor functions, for it is at that point when fresh state is allocated in both programs.

So, assuming we have been given a relational interpretation $\chi_\alpha \in \mathit{Type}[\tau_\alpha,\tau'_\alpha]$ for the type parameter α, consider the proof that the constructor functions are logically related. When the constructors are called, we allocate fresh state: l in the first program, and $\langle l'_0,\langle l'_1,l'_2\rangle\rangle$ in the second program. For convenience, we will package these together notationally as $ls = \langle l,\langle l'_0,\langle l'_1,l'_2\rangle\rangle\rangle$. We now extend the current world W with w_{p+1}, defined as follows:

$$\begin{aligned}
w_{p+1} &= (\eta_{k_0}^{ls}, \mathcal{L}_{k_0}^{ls})\\
\eta_k^{ls} &= (\psi_k^{ls}, \{ls\}, \{l:\tau_\alpha\}, \{l'_0:\mathsf{int},\ l'_1:\tau'_\alpha,\ l'_2:\tau'_\alpha\})\\
\psi_k^{ls} &= \{(j,W,s,s') \in \mathit{StoreAtom}_k \mid\\
&\qquad \exists i\in\{1,2\}.\,s'(l'_0) = i \wedge (j,W,s(l),s'(l'_i)) \in \chi_\alpha\}\\
\mathcal{L}_k^{ls} &= \{(j,\eta_j^{ls}) \mid j\le k\}
\end{aligned}$$

The store relation ψ_k^{ls} ensures that the contents of l are related (by χ_α) to the contents of the proper slot l'_1 or l'_2, depending on the current flag value stored in l'_0. Note how the definition of ψ_k^{ls} relies crucially on the presence of the world parameter W. Without it, we would not know in which world to compare $s(l)$ and $s'(l'_i)$. Note also that in this example w_{p+1} does not evolve (*i.e.*, its store relation remains the same in all future worlds).

Finally, when proving equivalence of the existential packages, we represent the cell class type β with χ_β defined as follows:

$$\begin{aligned}
\chi_\beta = \{(j,W,l,\langle l'_0,\langle l'_1,l'_2\rangle\rangle) \mid\ & W\in\mathit{World}_j \wedge\\
& \exists w\in W.\,w = (\eta_j^{ls},\mathcal{L}_j^{ls}),\ \text{where } ls = \langle l,\langle l'_0,\langle l'_1,l'_2\rangle\rangle\rangle\}
\end{aligned}$$

Note that χ_β includes ls's owned by *any* island of the right form. This might add some "junk" to the relation (*e.g.*, objects that were created by some other class's constructor function), but any such junk is harmless since it adheres to the same invariants that the objects created by e and e' do.

5.5 Irreversible State Changes

A well-known example that has caused trouble for previous logical relations methods is Pitts and Stark's "awkward" example [21]. Although this example does not involve existentials, it has proven difficult to handle because it involves an *irreversible state change*:

$$\begin{aligned}
e &= \mathtt{let}\,x = \mathtt{ref}\,0\,\mathtt{in}\,\lambda f:(\mathsf{unit}\to\mathsf{unit}).\,(x\mathtt{:=}1; f(); \mathtt{!}x)\\
e' &= \lambda f:(\mathsf{unit}\to\mathsf{unit}).\,(f(); 1)
\end{aligned}$$

The idea here is that e and e' are equivalent because, as soon as they are applied, the contents of x are set to 1, after which point $\mathtt{!}x$ will always return 1. In other words, the first application of e marks an irreversible state change from $x\mapsto 0$ to $x\mapsto 1$.

Intuitively, irreversible state changes are hard to handle if the knowledge about a piece of local state is fixed once and for all at the point it is allocated. Using traditional possible-worlds models, the most precise invariant one can enforce about the contents of x is that they are *either* 0 or 1. With such a weak invariant, it is impossible to know when returning from $f()$ whether $\mathtt{!}x$ is still 1.

Using populations, however, we can prove the equivalence of e and e' quite easily. A suitable island definition is:

$$\begin{aligned}
w_{p+1} &= (\eta_{k_0}^{\emptyset}, \mathcal{L}_{k_0})\\
\eta_k^V &= (\psi_k^V, V, \{l_x:\mathsf{int}\}, \{\})\\
\psi_k^V &= \{(j,W,s,s') \in \mathit{StoreAtom}_k \mid s(l_x) = |V|\}\\
\mathcal{L}_k &= \{(j,\eta_j^V) \mid j\le k \wedge |V|\le 1\}
\end{aligned}$$

The intuition here is that we use V to encode a flag telling us whether x has already been set to 1. Initially, $\mathtt{!}x$ is 0, signified by $V=\emptyset$. When x is set to 1, we add some arbitrary value to V, making it a singleton set of size 1. Because V is only allowed to grow, we know that x can never be changed back to 0. In addition, since the law $\mathcal{L}_k$ requires $|V|\le 1$, x must remain at 1 permanently.

5.6 Callback with Lock

The proofs for the examples presented so far do not use step indices in an interesting way. The last of our examples, which is inspired by the reentrant callback example of Banerjee and Naumann [6], demonstrates an unexpected case where the steps come in handy. Relying as it does on subtle stepwise reasoning, our proof for this example is rather involved (some might say ugly), but like a dog walking on its hind legs, one is surprised to find it done at all.

Consider the following object encoding of higher-order type $\tau = ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{int})$:

$$e \quad = \quad C[f\,(); x := !x + 1]$$

where

$$C \quad = \quad \mathtt{let}\, x = \mathtt{ref}\, 0\, \mathtt{in}\, \langle \lambda f : \mathsf{unit} \to \mathsf{unit}.\, [\cdot], \lambda z : \mathsf{unit}.\, !x \rangle$$

It implements a counter object with two methods: an increment function, and a get function requesting the current counter value. An interesting feature of this object is that its increment method takes a callback argument, which is invoked before the counter is incremented.

Now, consider the following alternative implementation for this object, in which x is dereferenced *before* the callback:

$$e' \quad = \quad C[\mathtt{let}\, n = !x\, \mathtt{in}\, f\,(); x := n + 1]$$

One might naïvely assume that the two versions are equivalent, because x is not publicly accessible. But f might perform arbitrary operations, including recursively calling the increment function! In this case, x may be modified between read and write access in e'.

Such reentrance can be prevented by adding a lock:

$$\begin{array}{l} C = \mathtt{let}\, b = \mathtt{ref}\, \mathtt{true}\, \mathtt{in} \\ \quad \mathtt{let}\, x = \mathtt{ref}\, 0\, \mathtt{in} \\ \quad \langle \lambda f : \mathsf{unit} \to \mathsf{unit}. \\ \qquad (\mathtt{if}\, !b\, \mathtt{then}\, (b := \mathtt{false}; [\cdot]; b := \mathtt{true})\, \mathtt{else}\, ()) \\ \quad \lambda z : \mathsf{unit}.\, !x \rangle \end{array}$$

Note that it is still possible for f to invoke the get function, which just *reads* the current x.

With C reimplemented using a lock, e and e' are now contextually equivalent. But how do we go about actually proving this? To show the two increment functions equivalent, we need to establish that f cannot modify x. But how can we set up an island that ensures that? After all, the island's law must certainly allow updates to x *in general*. How can we formulate a law that allows the store to change, but still can *temporarily* prohibit it?

Steps to the rescue! When proving that the two increment functions are related, we assume that one terminates with j steps. Assuming b is set to `true` (*i.e.*, assuming that x is "unlocked"), we can partition the reduction sequence for its execution into 3 phases of length $j_1 + j_2 + j_3 = j$, where j_2 spans the steps spent in the call to f. These j_2 steps are the time window in which x is not allowed to change. So the idea is to define a law that allows setting up time windows of this kind, during which $!x$ must remain constant.

The following island definition does the trick:

$$\begin{array}{rl} w_{p+1} = & (\eta_{k_0}^{\{\langle k_0, k_0, 0\rangle\}}, \mathcal{L}_{k_0}) \\ \eta_k^V = & (\psi_k^{\min(V)}, V, \{l_b : \mathsf{bool}, l_x : \mathsf{int}\}, \{l'_b : \mathsf{bool}, l'_x : \mathsf{int}\}) \\ \psi_k^{\langle k_1, k_2, n\rangle} = & \{(j, W, s, s') \in \mathit{StoreAtom}_k \mid \\ & \quad (j \le k_1 \wedge s(l_b) = s'(l'_b) \wedge s(l_x) = s'(l'_x)) \wedge \\ & \quad (j \ge k_2 \Rightarrow (s(l_b) = \mathtt{false} \wedge s(l_x) = n))\} \\ \mathcal{L}_k = & \{(j, \eta_j^V) \in \mathit{LawAtom}_k \mid \\ & \quad V = \{\langle k_1, k'_1, n_1\rangle, \ldots, \langle k_m, k'_m, n_m\rangle\} \wedge \\ & \quad k_1 \ge k'_1 > k_2 \ge \cdots \ge k'_{m-1} > k_m \ge k'_m\} \end{array}$$

Each window is represented by a triple $\langle k_1, k_2, n\rangle$ in V (assuming the obvious encoding of triples using pairs), with k_1 and k_2 giving its first (upper) and last (lower) step, and n being the value to which x is fixed during the window. The side condition in $\mathcal{L}_k$ ensures that windows do not overlap. Consequently, there is always a unique lowest (newest) window $\min(V) = \langle k_1, k_2, n\rangle$, *i.e.*, the one with the least first projection (the k_1). The store relation ψ ensures that, if the step level j has not yet passed the lower bound k_2 of the newest window (*i.e.*, if $j \ge k_2$), then $!x$ must equal the n from that window, and the lock must be held. The definition of ψ also prohibits windows from starting in the future by requiring $j \le k_1$.

To prove equivalence of the increment functions, starting at step k with $s_0(l_x) = s'_0(l'_x) = n$ and $s_0(l_b) = s'_0(l'_b) = \mathtt{true}$ (the interesting case), we proceed in j_1 steps to set b to `false`, and then add a new lowest window $\langle k - j_1, k - j_1 - j_2 - 1, n\rangle$ to the population of the $(p{+}1)$-th island. Next, we know $f()$ returns after exactly j_2 steps in some future world W, and the stores s_1 and s'_1 that it returns must be related by W at step $m = k - j_1 - j_2$, which means that $(m - 1, \lfloor W \rfloor_{m-1}, s_1, s'_1) \in W[p+1].\psi$. Since the step level $m - 1$ is still in the range of the window we installed, we know that $f()$ could not have added an even lower window to the population of the $(p{+}1)$-th island (as the law disallows adding windows that start in the future). Thus, we know that $W[p{+}1].\psi = \psi_m^{\langle m+j_2, m-1, n\rangle}$, and consequently $s_1(l_x) = s'_1(l'_x) = n$ and $s_1(l_b) = s'_1(l'_b) = \mathtt{false}$. That is, thanks to our use of the lock, the call to $f()$ could not have affected our local state.

5.7 Well-Bracketed State Changes

To conclude, we give two examples that our method appears *unable* to handle. The first one, suggested to us by Jacob Thamsborg, is a variant of Pitts and Stark's "awkward" example (Section 5.5):

$$\begin{array}{rcl} e & = & \mathtt{let}\, x = \mathtt{ref}\, 0\, \mathtt{in} \\ & & \lambda f : (\mathsf{unit} \to \mathsf{unit}).\, (x := 0; f(); x := 1; f(); !x) \\ e' & = & \lambda f : (\mathsf{unit} \to \mathsf{unit}).\, (f(); f(); 1) \end{array}$$

Here, unlike in the "awkward" example, the state of x changes back and forth between 0 and 1. The reason we believe e and e' to be equivalent (we do not have a proof!) is that the state changes occur in a "well-bracketed" fashion — *i.e.*, every change to 0 is guaranteed to be followed later on in the computation by a change to 1. This implies (informally) that invoking the callback function f will either leave the state of x unchanged or will return control with x set to 1. However, it is not clear to us how to formally establish this. The trick of representing irreversible state changes via population growth is inapplicable since the state changes are not irreversible, and the time windows idea from Section 5.6 is inapplicable as well since the example does not make use of locks.

5.8 Deferred Divergence

Here is another example we cannot handle, due to Hongseok Yang:[2]

$$\begin{array}{rcl} e_1 & = & \lambda f : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}.\, f\,(\lambda z : \mathsf{unit}.\, \mathtt{diverge}) \\ e_2 & = & \lambda f : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}. \\ & & \quad \mathtt{let}\, x = \mathtt{ref}\, 0\, \mathtt{in}\, \mathtt{let}\, y = \mathtt{ref}\, 0\, \mathtt{in} \\ & & \quad f\,(\lambda z : \mathsf{unit}.\, \mathtt{if}\, !x = 0\, \mathtt{then}\, y := 1\, \mathtt{else}\, \mathtt{diverge}); \\ & & \quad \mathtt{if}\, !y = 0\, \mathtt{then}\, x := 1\, \mathtt{else}\, \mathtt{diverge} \end{array}$$

Here, f may either call its argument directly, in which case the computation clearly diverges (in e_2 this happens eventually because y is set to 1), or it may store its argument in some ref cell. In the latter case, any subsequent call to the stored argument by the program context will also cause divergence (in the case of e_2, because x will be 1 at that point). Only if neither f nor the context ever tries to call f's argument may the computation terminate.

[2] A similar example is discussed in Benton and Leperchey [7], at the end of their section 5. However, the two terms in their example are not actually equivalent in our language, because we have higher-order store.

For us to prove e_1 and e_2 equivalent, we would need some way of relating the two arguments to f. Initially, however, when the arguments are invoked, one terminates and the other does not, so it is not obvious how to relate them. In fact, they are only related under the knowledge of what e_1 and e_2 will do after the call to f. This suggests to us that one way to handle such an example might be to define a relation on terms coupled with their continuations.

6. Related and Future Work

There is a vast body of work on methods for reasoning about local state and abstract data types. In the interest of space, we only cite a representative fraction of the most closely related recent work.

Logical Relations Our work continues (and, to an extent, synthesizes) two lines of recent work: one on using logical relations to reason about type abstraction in more realistic languages, the other on using logical relations to reason about local state.

Concerning the former, Pitts [20] provides an excellent overview, although it is now slightly outdated — in the last few years, several different logical relations approaches have been proposed for handling general recursive (as well as polymorphic) types [16, 1, 11], which Pitts considers an open problem. Much of the work on this topic is concerned with logical relations that are both sound *and complete* with respect to contextual equivalence. Completeness is useful for establishing various *extensionality properties* at different types, *e.g.*, that two values of type $\forall\alpha.\tau$ are contextually equivalent iff their instantiations at any particular type τ' are equivalent. In general, however, just because a method is complete with respect to contextual equivalence does not mean that it is *effective* in proving all contextual equivalences. In fact, Pitts gives a representation independence example for which existing techniques are "effectively" incomplete.[3]

For a logical relation to be complete it must typically be what Pitts terms "equivalence-respecting." There are different ways to achieve this condition, such as $\top\top$-closure [20], biorthogonality [16], or working with contextual equivalence classes of terms [11]. Pitts' $\top\top$-closure neatly combines the equivalence-respecting property together with *admissibility* (or *continuity*, necessary for handling recursive functions) into one package.

We build on the work of Ahmed [1] on step-indexed logical relations for recursive and quantified types. One advantage of the step-indexed approach is that admissibility comes "for free," in the sense that it is built directly into the model. By only ever reasoning about finite approximations of the logical relation ($\mathcal{V}_n[\![\tau]\!]\,\rho$), we avoid the need to ever prove admissibility. (In other words, an inadmissible relation is indistinguishable from an admissible one if one only ever examines its step-indexed approximations.) Of course, the price one pays for this is that one is forced to use stepwise reasoning *everywhere*, so admissibility is not really "free" after all. To ameliorate this burden, we are currently investigating techniques for proving logical approximation in our model *without* having to do explicit stepwise reasoning. As we saw in Section 5.6, though, sometimes the presence of the step indices can be helpful.

Like Ahmed's previous work, our logical relation is sound, but not complete, with respect to contextual equivalence. (Hers is complete except for the case of existential types.[4]) While our method cannot in its current form prove extensionality properties of contextual equivalence, it is still useful for proving representation independence results, which is our primary focus. Recent work by Ahmed and Blume [2] involves a variant of [1] that *is* complete with respect to contextual equivalence, where completeness is obtained by essentially Church-encoding the logical interpretation of existentials (this is roughly similar to what $\top\top$-closure does, too). We are currently attempting to develop a complete version of our method, using a similar approach to Ahmed and Blume.

Concerning the second line of work — logical relations for reasoning about local state — most of the recent previous work we know of employs possible-worlds models of the sort we discussed in Section 3.2, so we refer the reader to that earlier section for a thorough comparison [21, 22, 7, 10]. However, there are two recent pieces of work that are worth discussing in further detail.

Perhaps the closest related work to ours is Nina Bohr's PhD thesis [9], which extends her work with Lars Birkedal [10] in two directions. First, she gives a denotational possible-worlds model for a language with general recursive types, polymorphism, and higher-order references, with the restriction that references must have *closed* type. This restriction seems to imply that her method is inapplicable to the cell class example in Section 5.4 because it involves references of type $\mathsf{ref}\,\alpha$. Second, she proposes a more refined (and complex) notion of possible world in which an island's store relation has the ability to change over time. This is similar in certain ways to our population technique, except that her islands do not contain anything resembling a population. Her approach is designed to handle examples involving irreversible state changes, like Pitts and Stark's "awkward" example (Section 5.5), but *not* generative ADTs (Sections 5.1–5.3). Bohr's possible worlds also include the ability to impose invariants on the *continuations* of related terms, so we believe her technique can handle at least one, if not both, of the examples in Section 5.7 and 5.8, which we cannot.

In a paper conceived concurrently with ours, Birkedal, Støvring, and Thamsborg [8] present a relationally parametric denotational model of a language with general recursive types, polymorphism, and references of arbitrary type. Their model improves on Bohr's in the flexibility of its references, but it offers only a weak notion of possible worlds, with which one can only do very simple reasoning about local state. Their model cannot handle any of our examples.

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 *bisimulations*. Pierce and Sangiorgi [19] define a bisimulation for reasoning about polymorphic π-calculus, and they demonstrate its effectiveness on an example that is similar to our symbol table example. Due to the low-level, imperative nature of the π-calculus, it is difficult to give a precise comparison between their technique and ours, but the basic idea of their technique (described below) has been quite influential on subsequent work.

Sumii and Pierce define bisimulations for an untyped language with a dynamic sealing operator [27], as well as an extension of System F with general recursive types [28]. Koutavas and Wand [13] adapt the Sumii-Pierce technique to handle an untyped higher-order language with general references; in the process, they improve on Sumii-Pierce's treatment of contextual equivalences involving higher-order functions. Interestingly, the Koutavas-Wand technique involves the use of inductive stepwise reasoning when showing that two functions are in the bisimulation. Subsequently, Sangiorgi, Kobayashi, and Sumii [26] propose *environmental bisimulations*, which generalize Sumii and Pierce's previous work to an untyped framework subsuming that of Koutavas-Wand's, but in a way that does not appear to require any stepwise reasoning. While all of these bisimulation approaches are sound and complete with respect to contextual equivalence, none handles a language with both existential type abstraction and mutable state.

[3] Pitts' example is actually provable quite easily by a *transitive* combination of logical relations proofs (`www.mpi-sws.org/~dreyer/pitts.txt`). Dreyer has suggested a harder example, mentioned on page 25 of Sumii and Pierce [28], for which there is not even any known "brute-force" proof.

[4] The published conference version of her paper claims full completeness, but the proof contains a technical flaw uncovered by the second author. The extended version of her paper corrects the error [1].

There are many similarities between bisimulations and logical relations, although a precise comparison of the techniques remains elusive (and an extremely interesting direction for future work). With bisimulations, one defines the relational interpretations of abstract types, or the invariants about local state, *up front*, as part of a relation also containing the terms one wishes to prove contextually equivalent, and then one proceeds to show that the relation one has defined is in fact a bisimulation. With logical relations, the proof proceeds backward in a structured way from the goal of showing two terms logically equivalent, and the invariants about type representations or local state are chosen in mid-proof. It is arguably easier to sketch a bisimulation proof (by just stating the bisimulation), whereas the islands and χ definitions in our proof sketches must be stated *in medias res*. On the other hand, our islands and χ's are more minimal than bisimulations, which must often explicitly include a number of redundant intermediate proof steps.

The Sumii-Pierce-Koutavas-Wand-Sangiorgi-Kobayashi-Sumii approach is roughly to define bisimulations as sets of relations, with each relation tied to a particular *environment*, *e.g.*, a type interpretation, a pair of stores, etc. Various "up-to" techniques are used to make bisimulations as small as possible. This approach seems conceptually similar to possible-worlds semantics, but the exact relationship is unclear, and we plan to explore the connection further in future work.

Separation Logic To reason about imperative programs in a localized manner, O'Hearn, Reynolds *et al.* introduced *separation logic* [24] as an extension to Hoare logic. Separation logic has been enormously influential in the last few years, but it has not to our knowledge been used to reason about higher-order typed functional languages with type abstraction and higher-order store. Notably, however, the desire to scale separation logic to reason about a functional programming language has led to Hoare Type Theory (HTT) [18]. HTT is a dependently typed system where computations are assigned a monadic type in the style of a Hoare triple. Under this approach, programs generally have to pass around explicit proof objects to establish properties. Currently, HTT only handles strong update (where a location's type can vary over time), not ML-style references with weak update (and thus stronger invariants).

Relational Reasoning About Classes There is a large body of work on reasoning techniques for object-oriented languages. For example, Banerjee and Naumann [5] present a denotational method for proving representation independence for a Java-like language. Koutavas and Wand [14] have adapted their bisimulation approach to a subset of Java. The languages considered in these works do not provide generativity and first-class existential types, but rather tie encapsulation to static class definitions. On the other hand, subsequent work by Banerjee and Naumann [6] addresses the issue of ownership transfer, which we do not. We believe that the generativity of existential quantification and the separation enforced by possible-island semantics are closely related to various notions of ownership and ownership types, but we leave the investigation of this correspondence to future work.

References

[1] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006. Extended/corrected version of this paper available as Harvard University TR-01-06.

[2] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ICFP*, 2008.

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence (Technical appendix), 2008. Available at: `http://ttic.uchicago.edu/~amal/papers/sdri/`.

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

[5] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

[6] Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In *ECOOP*, 2005.

[7] Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, 2005.

[8] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Relational parametricity for references and recursive types, July 2008. Draft, submitted for publication.

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

[10] Nina Bohr and Lars Birkedal. Relational reasoning for recursive types and references. In *APLAS*, 2006.

[11] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. In *Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin*. 2007.

[12] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL*, 2003.

[13] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, 2006.

[14] Vasileios Koutavas and Mitchell Wand. Reasoning about class behavior. In *FOOL/WOOD*, 2007.

[15] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, 1995.

[16] Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *LICS*, 2005.

[17] John C. Mitchell. Representation independence and data abstraction. In *POPL*, 1986.

[18] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5&6):865–911, September 2008.

[19] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000.

[20] Andrew Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 245–289. The MIT Press, 2005.

[21] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[22] Uday Reddy and Hongseok Yang. Correctness of data representations involving heap data structures. In *ESOP*, 2003.

[23] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, 1983.

[24] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[25] Claudio V. Russo. Non-dependent types for Standard ML modules. In *PPDP*, 1999.

[26] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *LICS*, 2007.

[27] Eijiro Sumii and Benjamin Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1–3):161–192, 2007.

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

[29] Janis Voigtländer and Patricia Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3):290–318, 2007.