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 pure extensions of System F, as well as in impure Java-like languages. However, none of the existing work on representation independence addresses the interaction of existential type abstraction and mutable state. Specifically, none shows how to prove equivalence of generative ADTs, for which each instance of the ADT must introduce a distinct abstract type because the relational interpretation of that type depends on some local instance-specific state.

In this work, we present a method for proving state-dependent representation independence. Our method extends Ahmed’s previous work on step-indexed logical relations for recursive and quantified types in order to handle ML-style unrestricted state. We use this method to prove a number of interesting contextual equivalences, some drawn from the literature on type generativity, that involve a close interaction between existentials and state. In these examples, the relational interpretations of the abstract types may be initially empty and then grow over time in a manner that is tightly coupled with changes to some local state. We encode such state-dependent type representations using a novel possible-worlds model that is inspired by several recent approaches but relies on step-indexing to enable a well-founded quasi-circular definition of possible worlds.

1. Introduction
Reynolds’ notion of relational parametricity [20] 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 [15] 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 last 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 [17], general recursive types [14, 2, 10], selective strictness [26], etc. Others have considered representation independence in an impure Java-like object-oriented setting, in which abstraction is achieved through nominal class typing and private access modifiers [6, 7, 13].

None of the existing work, however, studies representation independence in the context of a language supporting both existential type abstraction (as in System F) and mutable state (as in ML or Java). Why should one care about such a language?

1.1 Reasoning About Generative Type Abstraction
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 [22]. 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. [11]. (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 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) stack 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

[Copyright notice will appear here once 'preprint' option is removed.]
This is the case, for example, in OCaml, which only supports 1 valid in its own table a fresh symbol type. Each instantiation of the Symbol on the argument to S.t is replaced by:

```
val insert : string -> t
val eq : t * t -> bool
```

This kind of result can be understood as an instance of representation independence, albeit a somewhat degenerate one in 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 depends critically on step-indexing to support a quasi-circular definition of possible worlds.

The remainder of the paper is structured as follows. In Section 2, we present F^{µι}, 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 F^{µι} 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.

2. The Language F^{µι}

We consider F^{µι}, a call-by-value λ-calculus with impredicative polymorphism, iso-recursive types, and general ML-style references. The syntax of F^{µι} 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 (e1 == e2), but is otherwise standard.

We define a small-step operational semantics for F^{µι} 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.

\[ F^{µι} \text{ typing judgments have the form } \Delta ; \Gamma ; \Sigma \vdash e : \tau, \text{ where the contexts } \Delta, \Gamma, \text{ and } \Sigma \text{ are defined as in Figure 2.} \]

The type context \( \Delta \) is used to track the set of type variables in scope; the value context \( \Gamma \) is used to track the term variables in scope (along with their types \( \tau \), which must be well formed in context \( \Delta \), written \( \Delta \vdash \tau \)); and the store typing \( \Sigma \) tracks the types of the contents of locations in the store. Note that \( \Sigma \) maps locations to closed types. We write \( FTV (\tau) \) to denote the set of type variables that appear free in type \( \tau \). The typing rules are entirely standard, so we show only

---

1 This is the case, for example, in OCaml, which only supports applicative (i.e., non-generative) functors.
Types
\[ \tau ::= \alpha \mid \text{unit} \mid \text{int} \mid \text{bool} \mid \pi \times \tau_2 \mid \pi \rightarrow \tau_2 \mid \forall \alpha. \tau \mid \mu \tau \mid \ref \tau \]

Terms
\[ e ::= \text{true} \mid \text{false} \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \text{fst } e \mid \text{snd } e \mid \lambda x. \tau. e \mid e_1 \cdot e_2 \mid \text{pack } \tau, e \vdash \exists \alpha. \tau' \mid \text{unpack } e_1 \vdash \alpha, x, \tau \mid \text{fold } \tau \vdash \text{unfold } \ref \tau \mid \text{ref } e \mid e_1 \cdot e_2 \mid e_1 \equiv e_2 \]

Values
\[ v ::= \text{true} \mid \text{false} \mid \text{(0 | 1 | n)} \]

\[
\begin{array}{ll}
\text{Term} & \text{Value} \\
\Delta, \Gamma; \Sigma & \Delta; \Gamma; \Sigma \\
\alpha. \tau & \forall \alpha. \tau \\
\tau \rightarrow \tau & \tau \rightarrow \tau \rightarrow \tau \\
\mu \tau & \mu \tau \\
\ref \tau & \ref \tau \\
\text{ref } e & \text{ref } e \\
\end{array}
\]

### Definition 2.1 (Contextual Approximation & Equivalence)

Let \( \Delta; \Gamma; \Sigma \vdash e_1 : \tau \) and \( \Delta; \Gamma; \Sigma \vdash e_2 : \tau \).

\[
\begin{align*}
\Delta; \Gamma; \Sigma \vdash e_1 \equiv_{\text{ctx}} e_2 : \tau & \quad \text{def} \\
\forall \Delta', \Gamma'; \Sigma' \vdash C : (\Delta; \Gamma; \Sigma \vdash \tau) \Rightarrow (\Delta'; \Gamma'; \Sigma' \vdash \tau') & \quad \wedge s : \Sigma' \wedge s, C[e_1] \models \Rightarrow s, C[e_2] \models \\
\Delta; \Gamma; \Sigma \vdash e_1 \equiv_{\text{ctx}} e_2 : \tau & \quad \text{def} \\
\Delta; \Gamma; \Sigma \vdash e_1 \equiv_{\text{ctx}} e_2 : \tau & \quad \wedge \Delta; \Gamma; \Sigma \vdash e_2 \equiv_{\text{ctx}} e_1 : \tau
\end{align*}
\]

### 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 \( \bigvee [\tau] \rho \), which relates pairs of values at a type \( \tau \), where the free type variables of \( \tau \) are given relational interpretations in \( \rho \). The relation is “logical” in the sense that functions are defined to be related (at an arrow type) iff, when applied to related arguments, they return related results. We will show that this logical relation is sound with respect to contextual equivalence for \( F^{nl} \). This is useful because, for many examples, it is much easier to show that two programs are in the logical relation than to show that 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 \([18, 19, 8, 9]\). 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 \( \tau \) 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 \( \tau \) 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 \( \{ w_1, \ldots, w_n \} \), 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 the 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 \( w \).
ternal world $W_0$. Both functions allocate local state in the same way, namely by allocating one pointer for $\text{table}$ and one for $\text{size}$, so we will want to extend $W_0$ with an island $\text{sym}$ describing the local invariants on $\text{table}$ and $\text{size}$. How should we define $\text{sym}$?

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

We can also use $\text{sym}$ to enforce that $!\text{size}$ is the same in the stores of both programs, and similarly for $!\text{table}$. Unfortunately, while this is a necessary condition, it is not sufficient to prove that the range check on the argument of $\text{lookup}$ in the second Symbol implementation always evaluates to $\text{true}$. For that, we need a way of correlating the value of $!\text{size}$ and the possible values of type $\tau$, 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 $\psi$, store relations by $\tau$, and laws by $\mathcal{L}$.

Consider the Symbol example. Let us define $V_0 = \{1, \ldots, n\}$, and let $\psi_0$ be the store relation containing pairs of stores that obey the properties concerning $\text{table}$ and $\text{size}$ described in Section 3.2 and that, in addition, both map the location $\text{size}$ to $n$. The idea is that $V_0$ describes the set of values of type $\tau$, given that the current stores satisfy $\psi_0$. Thus, when we extend the initial world $W_0$ with an island $\text{sym}$ governing Symbol’s local state, we will choose that $\text{sym}$ to comprise population $V_0$, store relation $\psi_0$, and a law $\mathcal{L}$ defined as $\{\psi_0, V_0\} | n \geq 0$. Here, $V_0$ and $\psi_0$ characterize the initial world, in which there are no values of type $\tau$ 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 $n > 0$), but only in conjunction with stores that map $\text{size}$ 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 $\text{sym}$, we are now able to define a relational interpretation for the type $\tau$, namely: values $v_1$ and $v_2$ are related at type $\tau$ in world $W$ if $v_1 = v_2 = n$, where $n$ belongs to the population of $\text{sym}$ in $W$. In proving equivalence of the two versions of the $\text{lookup}$ 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 $W_0 \uplus \text{sym}$, and that the arguments to $\text{lookup}$ are related at type $\tau$ in $W$. Consequently, given the law that we established for $\text{sym}$ together with the interpretation of $\tau$, we know that the arguments to $\text{lookup}$ must both equal some $n$, that the current population of $\text{sym}$ must be some $V_0$, where $1 \leq n \leq m$, and that the current store relation must be $\psi_0$. Since $s_1$ and $s_2$ satisfy $W$, they must satisfy $\psi_0$, which means they map $\text{size}$ to $m \geq n$. Hence, the dynamic range check in the second version of Symbol must evaluate to $\text{true}$.

For the above relational interpretation of $\tau$ to make sense, we clearly need to be able to refer to a particular island in a world (e.g., $\text{sym}$) 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 $\psi$), we require that island populations can only grow larger in future worlds, not smaller.

### 3.4 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 [18], for example, only allow references to integers. Reddy and Yang [19] and Benton and Lepchey [8] additionally allow references to data, which include integers and pointers but not functions. While the exploration of relational reasoning in less expressive languages is certainly a worthwhile endeavor, in this work we are concerned with contextual equivalence in the presence of unrestricted state.

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

Bohr and Birkedal [9] 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 $(P, \{l_1, l_2, \tau\})$, 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 styled 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 $\psi$ in $W$, $(s_1, s_2) \in \psi(W)$, i.e., $s_1$ and $s_2$ obey $\psi$ 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 [5] 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 [2] 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 $V[I] \rho$ as the limit of an infinite chain of approximation relations $V[I] \rho \eta$, where $\eta \geq 0$. 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 fold $v_1$ and fold $v_2$ are defined to be related by $V[I] \rho \eta$ if $v_1$ and $v_2$ are related by $V[I] \rho \eta$ for all $\eta \geq 0$. 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 $v_1$ from fold $v_1$.

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 $\psi$’s) are parameterized by $(n-1)$-level worlds. The intuition behind this stratification of worlds is actually very simple: the knowledge behind this stratification of worlds is actually very simple: the $n$-level world describes properties of the current stores that may contain locations, but no free type or term variables). Given a set $\chi$ of this form, we write $\chi^m$ to denote the subset of $\chi$ such that $e_1$ and $e_2$ are values.

As mentioned in Section 3.3, a world $W$ is an ordered list (written $(w_1, \ldots, w_n)$) of islands. An island $w$ is a pair of some current knowledge $\eta$ and a law $L$. The knowledge $\eta$ for each island represents the current “state” of the island. It comprises four parts: a store relation $\psi$ (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 $\Sigma_1$ and $\Sigma_2$. The domains of $\Sigma_1$ and $\Sigma_2$ give us the sets of locations that the island “cares about” (a notion we mentioned in Section 3.2). Meanwhile, a law $L$ is a set of pairs $(k, \eta)$. If $k \in L$, it means that, at “time” $k$ (representing the number of steps left on the clock), the knowledge $\eta$ represents an acceptable state for the island. Below we summarize our notation for ease of reference.

$$\begin{align*}
\text{Type Interpretation} & \chi \define (k, W, e_1, e_2, \ldots) \\
\text{Store Relation} & \psi \define (k, W, s_1, s_2, \ldots) \\
\text{Population} & V \define \{v_1, \ldots\} \\
\text{Knowledge} & \eta \define (\psi, V, \Sigma_1, \Sigma_2) \\
\text{Law} & L \define \{(k, \eta)\ldots\} \\
\text{Island} & w \define (\eta, L) \\
\text{World} & W \define \{w_1, \ldots, w_n\}
\end{align*}$$

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

$$\begin{align*}
w.\eta & \define \eta_i \\
w.L & \define L_i \\
w.V & \define V_i \\
w.\psi & \define \psi_i \\
w.\Sigma_1 & \define \Sigma_{i1} \\
w.\Sigma_2 & \define \Sigma_{i2}
\end{align*}$$

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

$$\begin{align*}
\Sigma_1(W) & \define \cup_{1 \leq j \leq n} W[j].\Sigma_1 \\
\Sigma_2(W) & \define \cup_{1 \leq j \leq n} W[j].\Sigma_2
\end{align*}$$

We write $Val$ for the set of all values, $Store$ for the set of all stores (finite maps from locations to values), and $StoreTy$ for the set of store typings (finite maps from locations to closed types). We write $Population$ for the set of all subsets of $Val$. Finally, we write $S_1 \neq S_2$ 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 naively 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:

$$\begin{align*}
\text{StoreRel} & \define \mathcal{P}(\mathcal{N} \times \text{World} \times \text{Store} \times \text{Store}) \\
\text{World} & \define \text{StoreRel}^0
\end{align*}$$

A simple diagonalization argument shows that the set $\text{StoreRel}$ 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 $\text{CandAtom}_k$ and $\text{CandStoreAtom}_k$ 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 $\text{CandLawAtom}_k$ 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 these sets. We need some additional functions and predicates.

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

Next, we define the notion of world extension (see Figure 3, middle). We write $(j, W') \subseteq (k, W)$ (where $\subseteq$ is pronounced “extends”) if $W'$ is a world that is good for $k$ steps (i.e., $W \in \text{World}_k$, see below), $W'$ is a good world for $j \leq k$ steps ($W' \in \text{World}_j$), and $W'$ extends $[W]_j$ (written $W' \supseteq [W]_j$). 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 $w \in [W]_j$, the island $w'$ in the same position in $W'$ must extend $w$. Here we require that $w'.L = w.L$ 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 $W' \in \text{World}_j$ (which we discuss below) ensures that the future knowledge $w'.\eta$ obeys the law $w'.L$.
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 [17], our model is built from syntactically well-typed terms. Thus, we define \( \text{Type}[\tau_1, \tau_2]_k \) 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 \( \Sigma_1(W) \) denotes the “global” typing—i.e., the union of the \( \Sigma_1 \) components of all the islands in \( W \).) We also require the world \( W \) to be a member of \( \text{World}_1 \).

We define \( \text{Type}[\tau_1, \tau_2]_k \) as those sets \( \chi \subseteq \text{Atom}[\tau_1, \tau_2]_k \) that are closed under world extension. Informally, if \( v_1 \) and \( v_2 \) are related for \( j \) 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 \( (i, W) \). We define \( \text{StoreRel}_1 \) as the set of all \( \psi \subseteq \text{StoreAtom}_k \subseteq \text{CandAtom}_k \) that are closed under world extension. Again, the intuition for this property is that if two stores \( s_1 \) and \( s_2 \) are related for \( k \) steps in world \( W \), then they should continue to be related in all future worlds. This makes sense since stores are essentially tuples of values.

Knowledge \( \eta \) is the set of all tuples of the form \( (\psi, W, \Sigma_1, \Sigma_2) \) in \( \text{CandKnowledge} \) such that \( \psi \subseteq \text{StoreRel}_1 \). As mentioned above, the domains of \( \Sigma_1 \) and \( \Sigma_2 \) 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 \( \psi \), we cannot depend upon the contents of any location \( l \) in store \( s_1 \) that is not in \( \text{dom}(\Sigma_1) \) or on the contents of any location in \( s_2 \) that is not in \( \text{dom}(\Sigma_2) \). Thus, \( \Sigma_1 \) and \( \Sigma_2 \) essentially serve as accessibility maps [8]. While Benton and Leprechey’s accessibility maps are functions from store to subsets of \( \text{Loc} \), our accessibility maps are sets of locations that are allowed to grow over time.

We define \( \text{Law}_1 \) as the set of \( \zeta \) such that, for all \( (j, \eta) \in \zeta \), we have that \( \eta \in \text{Knowledge}_1 \). Furthermore, we require that the sets \( \zeta \) be closed under decreasing step index—that is, if some knowledge \( \eta \) obeys law \( \zeta \) for \( j \) steps, then it must be the case that any future time, when we have \( i < j \) steps left, the \( i \)-approximation of knowledge \( \eta \) still obeys the law \( \zeta \).

\( \text{Island}_d \) is the set of all pairs \( (\eta, \zeta) \in (\text{Knowledge}_1 \times \text{Law}_1) \) such that the knowledge \( \eta \) obeys the law \( \zeta \) at the current time denoted by step index \( k \)—i.e., \( (\eta, \zeta) \in \zeta \).

Finally, we define \( \text{World}_d \) as the set of all \( W \in (\text{Island}_d)^a \). 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 \( \text{F}^d \). The relations \( \mathcal{V}_n[\tau]_\rho \) are defined by induction on \( n \) and nested induction on the type \( \tau \). We use the metavariable \( \rho \) to denote type substitutions. A type substitution \( \rho \) is a finite map from type variables \( \alpha \) to triples \( (\chi, \tau_1, \tau_2) \) where \( \tau_1 \) and \( \tau_2 \) are closed types, and \( \chi \) is a relational interpretation in \( \text{Type}[\tau_1, \tau_2] \). If \( (\chi, \tau_1, \tau_2) \), then \( p_\alpha(\chi) \) denotes \( \chi \) and \( p_\alpha(\tau_1) \) denotes \( \tau_1 \).

Note that, by the definition of \( \mathcal{V}_n[\tau]_\rho \), if \( (k, W, e_1, e_2) \in \mathcal{V}_n[\tau]_\rho \), then \( k < n, W \in \text{World}_d, \Sigma_1(W) \vdash e_1 : p_\alpha(\tau), \) and \( \Sigma_2(W) \vdash e_2 : p_\alpha(\tau) \). Most of the relations \( \mathcal{V}_n[\tau]_\rho \) 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.

Two functions \( \lambda \cdot p_\chi(\tau).e_1 \) and \( \lambda \cdot p_\chi(\tau).e_2 \) are related for \( k \) steps in world \( W \) at the type \( \tau \) if in any future world \( W' \) where there are \( j < k \) steps left to execute (since beta-reduction consumes a step), and we have arguments \( v_1, v_2 \) that are related for \( j \) steps in world \( W' \) at the argument type \( \tau \), then the terms \( v_1/\chi \) and \( v_2/\chi \) are defined.

---

**Figure 4.** Step-Indexed Logical Relations for \( \text{F}^d \)
and $\alpha \vDash \alpha$ are logically related for $j$ steps in world $W'$ at the type $\tau$ (i.e., they are in the relation $E_{\alpha}[\tau] \rho$). Parameterizing over an arbitrary future world $W'$ is necessary here in order to ensure closure of the logical relation under world extension.

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 $\chi$ of $\alpha$, and the latter involves choosing an arbitrary relational interpretation $\chi$ of $\alpha$. The way the worlds are manipulated follows in the style of the other rules. The logical relation for $\forall \alpha. \tau$ 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.)

We say that two 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—their children are in the store relations of—all the islands in $W$ at all future times when $j < k$.

Any two locations related at a type $\tau$ 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 $\tau$. To enforce this requirement, we simply install a special island $\varepsilon$ that only cares about the one location $l_1$ in $s_1$ and the one location $l_2$ in $s_2$. Furthermore, $\varepsilon$ has an empty population and a law that says the population should remain empty in future worlds. Finally, the island's fixed store relation $\psi$ relates all stores $s_1$ and $s_2$ whose contents at locations $l_1$ and $l_2$, respectively, are related at type $\tau$ for $j < k$ steps. Here $j < k$ suffices since pointer dereferencing consumes a step (see Section 3.5).

The relation $E_{\alpha}[\tau] \rho$ defines when two terms are related. Two well-typed terms $e_1$ and $e_2$ are related for $k$ steps at the type $\tau$ in world $W$ if, given two starting stores $s_1$ and $s_2$ that are related for $k$ steps at world $W$, if $e_1$, $e_2$ steps to a configuration $s_1', v_1$ in $j < k$ steps then the following conditions hold. First, $s_2$, $e_2$ must step to some $s_2', v_2$ in any number of steps. Second, there must exist a world $W' \in \text{World}_{\alpha}[\rho]$ 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 $\tau$.

The definitions of logical approximation and equivalence for open terms are given at the bottom of Figure 4. Here we state some of the main properties of our logical relation. Further details of the meta-theory will appear in the online technical appendix [4].

**Lemma 4.1 (Logical Relations Are Valid Type Interpretations)**

If $\Delta \vDash \tau$ and $\rho \in D[[\tau]]$, then $V[\tau] \rho \in \text{Type}[\rho_1(\tau), \rho_2(\tau)]$.

**Theorem 4.2 (Fundamental Property)**

If $\Delta; \Gamma; \Sigma \vDash e : \tau$ then $\Delta; \Gamma; \Sigma \vDash e \spin E_{\rho} e : \tau$.

**Theorem 4.3 (Soundness w.r.t. Contextual Equivalence)**

If $\Delta; \Gamma; \Sigma \vDash e_1 \spin E_{\rho} e_2 : \tau$ then $\Delta; \Gamma; \Sigma \vDash e_1 \spin e_2 : \tau$.

## 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 [2]. 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 [4].

### 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:

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

where $\sigma = 2\alpha. (\lambda x. (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \text{bool})$ and $(+++x)$ abbreviates the expression $(x := 1 + x + 1; !x)$, and $\text{let}$ is encoded in the standard way (using function application). The package defines an abstract type $\alpha$ 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 $\alpha$ 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 stores the highest value that has been used so far. The intended invariant of this implementation is that no value of type $\alpha$ 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 $\text{true}$.

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 $\text{true}$. We only show the one direction, $\vdash e \spin E_{\rho} e' : \sigma$. The other direction is proven analogously.

Because the terms are closed, this only requires showing that $(k_0, W_0, e, e') \in E[\sigma][0]$ 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 $s_0, e \rightarrow_{k_1} s_1, v_1$ with $k_1 < k_0$. According to the definition of $E[\sigma][0]$, we need to come up with a reduction $s_0, e' \rightarrow_{k_2} s_1', v_1'$ and a world $W_1$ such that $(k_0 - k_1, W_1) \supseteq (k_0, W_0)$ and:

$$s_1, s_1', k_0 - k_1, W_1 \land (k_0 - k_1, W_1, v_1, v_1') \in V[\sigma][0]$$

By inspecting the definition of reduction, we see that

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

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

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

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 $\langle w_1, \ldots, w_p, w_{p+1} \rangle$ where:

$$w_{p+1} = (\eta_{k_0 - k_1}, \psi; k_0 - k_1) \quad \psi_{k_0 - k_1} = \eta_{k_0 - k_1}^\prime \quad \eta_{k_0 - k_1} = \{ j, W, s, s' \in \text{StoreAtom} | s(t) = s'(t') = n \}$$

The population $V_{\alpha}$ consists of all integers that are "valid" names in a world, i.e., not greater than the current value of $x$. We have to show $(k_0 - k_1, W_1) \supseteq (k_0, W_0)$ and $s_1, s_1' : k_0 - k_1 W_1$. Both are straightforward.
By definition of $V[\exists \alpha, \tau]$, we need to continue by giving a relation $\chi_\alpha \in \text{Type}[\text{int}, \text{int}]$, such that:

$$(k_0 - k_1, W_1, \lambda_\alpha.(+1), \lambda_\alpha.(+1)) \in V[\text{unit} \rightarrow \alpha] \times (\alpha \rightarrow \text{bool})$$

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

$$\chi_\alpha = \{ (j, W, i, i) \in \text{Atom}[\text{int}, \text{int}] \mid k_0 \in W[p + 1], V \}$$

This interpretation of $\alpha$ depends on the (valid) assumption that it will only be considered at $W$‘s that are future worlds of $W_{\alpha}$ (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 $\chi_\alpha$, 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 $\alpha$ in worlds whose $(p + 1)$-th island population contains $i$, that is, where $\|i\| \geq i$.

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 $V[\tau \times \tau^\ast]$, it remains to be shown that:

1. $\{ k_0 - k_1, W_1, \lambda_\alpha.(+1), \lambda_\alpha.(+1) \} \in V[\text{unit} \rightarrow \alpha] \times (\alpha \rightarrow \text{bool})$
2. $\{ k_0 - k_1, W_1, \lambda_\alpha.(\leq t), \lambda_\alpha.(\leq t) \} \in V[\alpha \rightarrow \text{bool}]$

For each of these, we assume we begin in some strictly future world $W_{\alpha}$ in which $(k_2, W_2) \supseteq (k_0 - k_1, W_1)$ and $s_2, s_3 - k_2 W_2$.

First consider (1). We are given $s_2, (++1) \rightarrow^{k_2} s_3, v_3$ for some $k_3 < k_2$, and it remains to show that $s_2, (++1) \rightarrow^{k_3} s_3, v_3'$, such that $s_3, s_4$ and $v_3, v_3'$ are related in some future world $W_3$ such that $(k_2 - k_3, W_3) \supseteq (k_2, W_2)$.

From $(k_3, W_3) \supseteq (k_0 - k_1, W_1)$ we know that $W_2[p + 1], L = [W_1[p + 1], L][k_2] = [L][k_2]$. From that $(k_2, W_2[p + 1], \eta) \in [L][k_2]$ follows, and hence there exists $n$ such that $W_2[p + 1], \eta = \eta_{k_2}^n$. That is, $W_2[p + 1], \psi = \psi_{k_2}^n$ and $W_2[p + 1], V = V_{\eta}$. From $s_2, s_3 - k_2 W_2$ and $k_3 < k_2$ we can conclude $(k_3, [W_2][k_3], s_3, s_2') \in \psi_{k_3}$ and thus $s_2(t) = s_3(t') = n$. Consequently, $v_3 = v_3' = n + 1$, $s_3 = s_2[t \mapsto n + 1]$, and $s_3' = s_2[t' \mapsto n + 1]$.

Now we choose $W_3$ to be $W_2$ with its $(p + 1)$-th island updated to $(\eta_{k_2}^{n + 1} - k_3, L_{k_3} - k_3)$. Again, we have to check the relevant properties, $(k_2 - k_3, W_3) \supseteq (k_2, W_2)$ and $s_3 - k_3 - k_3 - k_3 W_3$, which are straightforward. Last, we have to show that the results $v_3, v_3'$ are related in $V[\alpha_\eta]$, under this world, i.e., $(k_2 - k_3, W_3[n + 1, n + 1]) \in \chi_\alpha$. Since $(n + 1, n + 1) \in V_{n + 1} = W_{3}[p + 1], V$, this is immediate from the definition of $\chi_\alpha$.

Now consider (2). The proof is similar to that for part (1), but simpler. We are given that $(k_2, W_2, v_2, v_2') \in V[\alpha_\eta] \rho = \chi_\alpha$, and $s_2, (\leq t) \rightarrow^{k_2} s_3, v_3$ for some $k_3 < k_2$. The main thing to show is that $v_2 = v_2'$ (we can pick the end world $W_3$ to just be $W_2$). As in part (1), we can reason that $W_2[p + 1], \eta = \eta_{k_2}^n$ for some $n$, and therefore that $s_2(l) = n$ and, by definition of $\chi_\alpha$, 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.

$$e = \text{pack ref unit}, (\lambda z : \text{unit}. (\lambda f : \text{ref unit}. (\lambda p : \text{snd p}) as \sigma \text{ where } \sigma = \exists \alpha. (\text{unit} \rightarrow \alpha) \times (\alpha \rightarrow \text{bool}).)$$

Here are a suitable island definition and type interpretation for $\alpha$:

$$w_{p+1} = (\eta_{k_0}^{n+1}, L_{k_0})$$

Here, and in the examples that follow, $k_0$ represents the current step level, and $p$ the number of islands in the current world $W_{\alpha}$, 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 $\{l_1, \ldots, l_m\}$ 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 $L_0$ is flexible enough to permit any partial bijection between $\{1, \ldots, n\}$ and Loc to evolve over time. We (ab)use term-level pairs $(l, i)$ to encode this partial bijection in $V$. This is sufficient to deduce $e = e'$ iff $l_1 = l_2$ 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 $\text{pack} \tau_1, \tau_2, e$ as $\exists \alpha, \beta, \sigma$ to abbreviate two nested existentials in the obvious way):

$$e = \text{let } x = \text{ref 0 in}$$

pack int, int; \lambda z : \text{unit}. (\lambda x : \text{int}. \lambda p : \text{int} \rightarrow \text{int}). \lambda p : (\text{int} 	imes \text{int}). (\lambda f : \text{snd f}) as \sigma$$

where $\sigma = \exists \alpha, \beta. (\text{unit} \rightarrow \alpha) \times (\text{unit} \rightarrow \beta) \times (\alpha \rightarrow \beta \rightarrow \text{bool})$.

Here we use a single counter to generate names of two types, $\alpha$ and $\beta$, 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 $\alpha$ or of type $\beta$ but not of both). The example is interesting, however, in that we have no way of knowing the interpretations of $\alpha$ and $\beta$ 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 $e'$ where the comparison operator is replaced by $\lambda p : (\text{int} \times \text{int}). \text{false}$.

The following $\sigma$ and $\chi$ definitions enable such a proof:

$$w_{p+1} = (\eta_{k_0}^{n+1}, L_{k_0})$$

$$\eta_{k_0} = (\psi_{k_0}^n, V_{0, S}, \{1 : \text{int}, \{l' : \text{int}\})$$

$$\psi_{k_0} = \{ (j, W, s, l) \in \text{StoreAtoms}_k | s(l) = s'(l') = n \}$$

$$V_{0, S} = \{ (1, i) | i \in S \} \cup \{ (2, i) | i \in \{1, \ldots, n\} \}$$

$$L_k = \{ (j, n_i S) | S \subseteq \{1, \ldots, n_i\} \}$$

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

$$\chi_\beta = \{ (j, W, i, i) \in \text{Atom}[\text{int}, \text{int}] \mid \{2, i\} \in W[p + 1], V \}$$

The population here is partitioned into the valid names for $\alpha$ and the valid names for $\beta$, 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 $\alpha$, 2 for $\beta$). When proving equivalence of the two comparison operators, the definitions of $\chi_\alpha$, $\chi_\beta$ 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 [12]:

e = \Lambda \alpha. \text{pack} \alpha \cdot \langle \lambda x : \alpha. \text{ref} \ x, \\
\quad \lambda r : \text{ref} \alpha. \text{tr}, \\
\quad \lambda (r, x) : \alpha \times \alpha. (r = x) \rangle \alpha \sigma

where \( \sigma = \exists \beta. (\alpha \rightarrow \beta) \times (\beta \rightarrow \alpha) \times (\beta \times \alpha \rightarrow \text{unit}) \). We use pattern matching notation here merely for clarity and brevity (imagine replacing occurrences of \( r \) and \( x \) in the third function with \text{fst} and \text{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 \( \alpha \), 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 \text{class}, where \( \beta \) 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 [13], but it is modeled with a Java-like nominal type system, not with existential types.)

Similar to [12], we want to prove this canonical cell implementation equivalent to one using two alternating slots:

\[ e' = \Lambda \alpha. \text{pack} (\text{ref} \times (\text{ref} \alpha \times \text{ref} \alpha)), \]

\[ (\lambda x : \alpha. (\text{ref} 1, \text{ref} \ x, \text{ref} \ x), \\
\quad \lambda (r, x) : (\text{ref} \times (\text{ref} \alpha \times \text{ref} \alpha)). (r = x) \rangle \alpha \sigma \]

The idea for this proof has to relate the objects created on each side. More precisely, it relates the associated locations, in the form of tuples \((l, (l_0, (l_1, l_2)))\), where \( l \) is the reference used by \( e \), while the others belong to the respective object from \( e' \). As in Section 5.2, we abuse term-level tuples to express the relation in the population \( V \). Assuming we have given a relational interpretation \( \chi_{\alpha} \in \text{Type}[\tau_{\alpha}, \tau_{\alpha}'] \) for the type parameter \( \alpha \), we define:

\[ w_{p+1} = \langle \eta_{k_0}, \Sigma_{k_0} \rangle \]

\[ \eta_{k'} = \langle \psi_{k} \rightarrow V, \psi \rightarrow \Sigma_{k'}, \psi \rightarrow \Sigma_{k} \rangle \]

\[ \psi_{k'} = \langle \langle j, W, s, s' \rangle \in \text{StoreAtoms}_k | \\
\quad \forall (l, (l_0, (l_1, l_2))) \in V. \exists i \in \{1, 2\}, \\
\quad s'(l'_i) = s'(l'_i) \rangle 
\]

\[ \Sigma_{k'} = \langle \langle l : \tau_{\alpha} | (l, (l_0, (l_1, l_2))) \in V \rangle \rangle \]

\[ \Sigma_{k'} = \langle \langle l : \text{int}, l_0' : \tau_{\alpha}, l_1' : \tau_{\alpha}, l_2' : \tau_{\alpha} | (l, (l_0', (l_1', l_2'))) \in V \rangle \rangle \]

\[ \chi_{\beta} = \langle \langle j, W, l, (l_0', (l_1', l_2')) \rangle \text{Atom} | \\
\quad \text{ref} \tau_{\alpha} \times \text{ref} \tau_{\alpha} \times \text{ref} \tau_{\alpha} \rangle_{k_0} \rangle_{k_0} | \\
\quad \langle l, (l_0', (l_1', l_2')) \rangle \in \text{Store}[p+1, W] \rangle \]

Here, we assume that \( V \) mentions any location at most once. The store relation \( \psi_{k'} \) ensures that, for each pair of objects recorded in \( V \), the content of its reference \( l \) in \( e \) is related to the proper slot \( l'_i \) or \( l'_0 \) in \( e' \), depending on the current flag value stored in its \( l'_0 \). Note how \( \psi_{k'} \) relies crucially on the presence of the world parameter \( W \).

5.5 Callback with Lock

The proofs for the examples presented so far do not use step indices in an interesting way. While it does not actually involve existential quantification, the last of our examples demonstrates an unexpected case where the steps come in handy. Our example is similar to the reentrant callback example of Banerjee and Naumann [7].
To prove equivalence of the increment functions, starting at step \(k\) with \(s(l_k) = s'(l'_k) = n\) and \(s(l_k) = s'(l'_k) = \text{true}\) (the interesting case), we proceed in \(j_1\) steps to set \(b\) to false, and then add a new lowest window \((k - j_1, k - j_2, n)\) 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 \(s\) and \(s'\) that it returns must be related by \(W\) at step \(m = k - j_1 - j_2\), which means that \((m - 1, [W]_{m-1}, s, s') \in W[p+1]_\psi\). 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 \(W[p+1]_\psi = \psi^{(m+1-j_2-m-1,n)}\), and consequently it is still the case that \(s(l_k) = s'(l'_k) = n\) and \(s(l_k) = s'(l'_k) = \text{false}\). That is, thanks to our use of the lock, the call to \(f()\) could not have affected our local state.

5.6 Deferred Divergence

To conclude, here is one interesting example, suggested to us by Hongseok Yang, that our method cannot handle:2

\[
\begin{align*}
\epsilon_1 &= \lambda f : \text{unit} \to \text{unit}. f (\lambda z : \text{unit}. \text{diverge}) \\
\epsilon_2 &= \lambda f : \text{unit} \to \text{unit}. \\
&\quad \text{let x = ref 0 in let y = ref 0 in} \\
&\quad f (\lambda z : \text{unit}. \text{if } !x = 0 \text{ then } y = 1 \text{ else } \text{diverge}) \\
&\quad \text{if } !y = 0 \text{ then } x = 1 \text{ else } \text{diverge}
\end{align*}
\]

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.

For us to prove \(\epsilon_1\) and \(\epsilon_2\) 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 \(\epsilon_1\) and \(\epsilon_2\) 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. For space reasons, we only cite a representative fraction of the most closely related recent work, but there are numerous other papers that we consider relevant as well. We encourage the reader to let us know what citations we have omitted that they feel are critically important.

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 [17] provides an excellent overview, although it is now slightly out-of-date — in the last few years, several different logical relations approaches have been proposed for handling general recursive (as well as polymorphic) types [14, 2, 10], 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 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 \(\gamma\) 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 \(\perp\perp\)-closure [17], biorthogonality [14], or working with contextual equivalence classes of terms [10]. Pitts’ \(\perp\perp\)-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 [2] 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 \((\mathcal{V}_n[^T]\rho)\), 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 stepwise reasoning. As we showed in Section 5.5, 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 [3] involves a variant of [2] 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 \(\perp\perp\)-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 detailed comparison with previous approaches [18, 19, 8, 9].

Recently, Acar et al. [1] gave an untyped relational step-indexed model for reasoning about the consistency of imperative self-adjusting computation. The model employs a simple notion of possible worlds for reasoning about accessibility of memory locations, but the worlds are restricted to only allow partial bijections between locations.

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. Sumii and Pierce define bisimulations for an untyped language with a dynamic sealing operator [24], as well as an extension of System F with general recursive types [25]. Koutavas and Wand [12] adapt the Sumii-Pierce technique to han-

---

2A similar example is discussed in Benton and Leperchey [8], 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.

3As it turns out, Pitts’ example is provable quite easily by a transitive combination of logical relations proofs (ttiv-c.org/dreyer/pitts.txt). Dreyer has suggested a harder example, mentioned in Sumii and Pierce [25] (page 25), for which there is not even any known “brute-force” proof.

4The 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 [2].
dle 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, thus leading Koutavas to dub his technique inductive bisimulations. Subsequently, Sangiorgi et al. [23] 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.

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 [21] 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) [16]. 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 [6] present a denotational method for proving representation independence for a Java-like language. Koutavas and Wand [13] 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. 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