Abstract

We give a denotational semantics for a region-based effect system that supports type abstraction in the sense that only externally visible effects need to be tracked: non-observable internal modifications, such as the reorganisation of a search tree or lazy initialisation, can count as ‘pure’ or ‘read only’. This ‘fictional purity’ allows clients of a module to validate soundly more effect-based program equivalences than would be possible with previous semantics. Our semantics uses a novel variant of logical relations that maps types not merely to partial equivalence relations on values, as is commonly done, but rather to a proof-relevant generalisation thereof, namely setoids. The objects of a setoid establish that values inhabit semantic types, whilst its morphisms are understood as proofs of semantic equivalence. The transition to proof-relevance solves two awkward problems caused by naïve use of existential proof of semantic equivalence. The transition to proof-relevance makes setoids inhabitable semantic types, whilst its morphisms are understood as proofs of semantic equivalence. The transition to proof-relevance makes setoids inhabitable semantic types, whilst its morphisms are understood as proofs of semantic equivalence.

We give a denotational semantics for a region-based effect system that supports type abstraction in the sense that only externally visible effects need to be tracked: non-observable internal modifications, such as the reorganisation of a search tree or lazy initialisation, can count as ‘pure’ or ‘read only’. This ‘fictional purity’ allows clients of a module to validate soundly more effect-based program equivalences than would be possible with previous semantics. Our semantics uses a novel variant of logical relations that maps types not merely to partial equivalence relations on values, as is commonly done, but rather to a proof-relevant generalisation thereof, namely setoids. The objects of a setoid establish that values inhabit semantic types, whilst its morphisms are understood as proofs of semantic equivalence. The transition to proof-relevance solves two awkward problems caused by naïve use of existential quantification in Kripke logical relations, namely failure of admissibility and spurious functional dependencies.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features – Dynamic storage management; F.3.2 [Logic and Meanings of Programs]: Semantics of Programming Languages – Denotational semantics, Program analysis; F.3.2 [Logic and Meanings of Programs]: Studies of Program Constructs – Type structure

General Terms Languages, Theory

Keywords Type and effect systems, region analysis, logical relations, parametricity, program transformation

1. Introduction

The last decade has witnessed significant progress in modelling and reasoning about the tricky combination of effects and higher-order language features (first-class functions, modules, classes). The object of study may be ML-like, Java-like, or assembly-like, but the common source of trickiness is the way effectful operations may be partially encapsulated behind higher-order abstractions. Several closely-related problems have been addressed using common techniques that include separation, Kripke logical relations and step-indexing. One is to devise models and reasoning principles for establishing contextual (in)equivalences [18, 36]. A second is to establish equivalence between high-level and low-level code fragments, e.g., for compiler correctness [5, 23]. A third is to define Hoare-style logics for showing programs satisfy assertions [38]. A fourth, which we address here, is to study type systems and analyses that can characterize particular classes of behavior (such as purity) and be used to justify equivalences more generically.

Effect systems [22] refine conventional types by adding information capturing an upper bound on the effects expressions may have. Several recent papers [3, 6, 7, 26, 41], have explored the semantics of effect systems, with a focus not merely on showing correctness of analyses, but on providing a rigorous account of the conditions under which effect-dependent optimizations and refactorings are sound. An example is the commutation of stateful computations M and N, subject to the condition that the sets of storage locations potentially written by M and N are disjoint, and that neither potentially reads a location that the other writes.

We seek interpretations of effect-refined types (over an unreined model) that can justify such equivalences. Indeed, it is the interpretations, rather than rules for assigning such types to terms, that we regard as primary. Types provide a common interface language that can be used by clients in modular reasoning about rewrites; types can be assigned to particular terms by a mixture of more or less sophisticated inference systems, or by deeper semantic reasoning about particular implementations.

A key notion in reasoning about encapsulated state is that of separation: invariants depending upon mutually disjoint parts of the store. Intuitively, if each function with direct access to a part preserves the corresponding invariant, then all the invariants will be preserved by any composition of functions. Disjointness is naïvely understood in terms of sets of heap locations. A memory allocator, for example, guarantees that its own private data structures, memory previously handed out to clients, and any freshly-allocated block inhabit mutually disjoint sets of locations. Since the introduction of fractional permissions, work on separation logic often goes beyond this simple model, introducing resources that are combined with a separating conjunction, but which are not literally interpreted as predicates on disjoint locations. Research on ‘domain-specific’ [28], ‘fictional’ [17, 25], ‘subjective’ [30], or ‘superficial’ [29] separation logics and type theories aims to allow custom notions of separable resource to be introduced and combined modularly. This paper presents a semantics for effect systems supporting fictional, or ‘abstract’, notions of both effects and separation.

We have previously interpreted effect-refined types for stateful computations as binary relations, defined via preservation of particular sets of store relations [7]. This already provides some abstraction. The semantics is extensional so, for example, a function that reads a reference cell but doesn’t produce observably different results depending on the value read can soundly be counted as pure (contrasting with, for example, models of permissions that instrument the concrete semantics). Such a semantics can also interpret the ‘masking’ rule, allowing certain non-observable effects not to appear in annotations [6]. But here we go further, generalizing the interpretation of regions to, intuitively, partial equiva-
version of its argument. Unlike previous models of effects, our model justifies the typing \( \text{memo} : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \), saying that if \( f \) is observationally pure, \( \text{memo} f \) is, and so can participate in any program equivalence relying on purity. For example, a client can deduce the observational equivalence of the programs below purely on the basis of typing:

\[
λty.\text{let } e = \text{memo} (λx.x + 1).\text{let } e = \text{memo} (λx.x + 1)\text{in } ty.e
\]

**Overlapping references** Let \( p, p' \) implement a bijection \( Z \rightarrow Z \times Z \), then

\[
\tau_{eq} := \text{unit} \rightarrow (\text{unit} \rightarrow \text{unit}) \times (\text{unit} \rightarrow \text{unit}) \times (\text{unit} \rightarrow \text{unit}) \times (\text{unit} \rightarrow \text{unit})
\]

expressing that it allocates in two regions, \( r1 \) and \( r2 \), and returns a tuple of functions, each of which reads or writes just one region. This typing justifies, for example, permuting writes to the two abstract references.

**Set factory** A more realistic example, for which we do not give concrete code, is a function \( \text{setfactory} \) that generates mutable integer sets. Calling \( \text{setfactory} \) allocates a new reference cell pointing to a linked list of integers (initially empty) and returns a triple of functions (\( \text{mem}, \text{add}, \text{rem} \)) for testing membership of the represented set, adding a new integer to the set and removing an integer from the set. The implementation of \( \text{mem} \) searches for its integer argument in the linked list, but at the same time mutates the list by removing duplicates and, just for fun, potentially relocating some of the nodes. The other operations can potentially perform similar optimizations.

We can justify the following semantic typing for \( \text{setfactory} \):

\[
\text{setfactory} : \forall.t.\text{unit} \rightarrow (\text{int} \rightarrow \text{bool}) \times (\text{int} \rightarrow \text{unit}) \times (\text{int} \rightarrow \text{unit})
\]

which expresses that \( \text{setfactory} \) allocates in some (possibly fresh) region \( r \) and returns operations that, observably, only read \( r \) (the first one) and/or write in \( r \) (the second and third one) even though, physically, all three functions may read, write, and allocate concrete locations.

Thus, these functions can participate in corresponding effect-dependent program equivalences. For example, \( \text{mem} \) operations may be swapped and duplicated.

**Lazy Initialization** The following definitions illustrate an ‘allocate-on-write’ form of lazy initialization:

\[
\begin{align*}
\text{make} & \quad = \quad λ_r.\text{ref}(0) \\
\text{getx} & \quad = \quad λp.\text{if } p = 0 \quad \text{then } 0 \quad \text{else } !p.1 \\
\text{gety} & \quad = \quad λp.\text{if } p = 0 \quad \text{then } 0 \quad \text{else } !p.2 \\
\text{setx} & \quad = \quad λp.\text{an}.\text{if } p = 0 \quad \text{then } p := (\text{ref}(n), \text{ref}(0)) \quad \text{else } !p.1 := n \\
\text{sety} & \quad = \quad λp.\text{an}.\text{if } p = 0 \quad \text{then } p := (\text{ref}(0), \text{ref}(0)) \quad \text{else } !p.2 := n
\end{align*}
\]

We can introduce a new abstract type \( \text{point}_{t1,t2} \) for which the following typings can be semantically justified:

\[
\begin{align*}
\text{make : unit} \rightarrow \text{point}_{t1,t2} \\
\text{getx : point}_{t1,t2} \rightarrow \text{int} \\
\text{gety : point}_{t1,t2} \rightarrow \text{int} \\
\text{setx : point}_{t1,t2} \rightarrow \text{unit} \\
\text{sety : point}_{t1,t2} \rightarrow \text{unit}
\end{align*}
\]

Observe that the allocation effects are flagged in the \( \text{make} \) function, even though the physical allocations happen on demand, in the set
functions. Note also that semantic reasoning is necessary to justify that these definitions are well-typed at all, as a point holds different kinds of value at different times.

3. Syntax and Semantics

In this section we define the syntax and denotational semantics of our untyped metalanguage for stateful computations. We also give an effect-refined type system for this language that is parameterized by semantically-justified axioms. We omit the standard details of interpreting CBV languages via the metalanguage and of adequacy, relating the operationally induced observational equivalence to equality in the model.

**Denotational Model** A predomain is an $\omega$-cpo, i.e., a partial order with suprema of ascending chains. A domain is a predomain with a least element, ⊥. Recall that $f : A \to A'$ is continuous if it is monotone $x \leq y \Rightarrow f(x) \leq f(y)$ and preserves suprema of chains, i.e., $f(\sup x_i) = \sup f(x_i)$. Any set is a predomain with the discrete order. If $X$ is a set and $A$ a predomain then any $f : X \to A$ is continuous. A subset $U$ of a predomain $A$ is admissible if whenever $(a_i)$ is an ascending chain in $A$ such that $a_i \in U$ for all $i$, then $a_i \in U$. If $f : X \times A \to A$ is continuous and $A$ is a domain then one defines $f(x) = \sup f(x_i)$ with $f_i(a) = f(x, a)$. One has, $f(x, f(x)) = f(x)$ and if $U \subseteq A$ is admissible and $f : X \times U \to U$ then $f : X \to U$, too. We denote a partial (continuous) function from set (predomain) $A$ to set (predomain) $B$ by $f : A \to B$.

We assume two sets $L$ and $V$, modelling concrete locations, and the $V$-values that can be stored in locations. We assume that $V$-values include integers, written $int(n)$ for some $n \in \mathbb{N}$, booleans, written $bool(b)$ for $b \in \mathbb{B}$, locations $loc(l)$ for some $l \in L$, and tuples of $V$-values, written $(v_1, ..., v_n)$. We assume that it is possible to tell whether a value is of that form and in this case to retrieve the components. Heaps $h \in H$ are finite maps from $L \to V_h$. The domain of a heap $h$ is $\text{dom}(h) \subseteq L$. We write $\emptyset h \in H$ for the empty heap. If $v \in V_h, l \in \text{dom}(h)$ then $h[l\mapsto v]$ is the heap that updates $l$ to contain $v$; $\text{new}(h, v)$ yields a pair $(l, h')$ where $l \in L$ and $h' \in H$.

The following properties hold: $\text{dom}(\emptyset) = \emptyset, \text{dom}(h[l\mapsto v]) = \text{dom}(h), (h[l\mapsto v])' = h[l' \mapsto v']$ if $v' = v$ and $\text{new}(h,v) = (l, h')$ then $\text{dom}(h') = \text{dom}(h) \cup \{l\}$ and $l \notin \text{dom}(h)$ and $h(l) = v$.

Note that heaps and $V$-values are discrete predomains; we do not here model higher-order store (storing objects with nontrivial order, such as functions). Our stores are ‘flat’, like those of object-oriented languages.

We define the predomain of values $V$ and the domain of computations $\mathcal{C}$ simultaneously as follows: $\mathcal{C} = H \times X \times V$ are partial continuous functions from $H \times X \times V$, the bottom element being the everywhere undefined function. Values are defined as either $R$-values, tuples of values or continuous functions from values to computations; $V = V_X + \text{fun}(V \to \mathcal{C}) + V^*$. Such domain equations can be solved by standard methods.

**Syntax** The syntax of untyped values and computations is:

$$v ::= x \mid 0 \mid c \mid (v_1, v_2) \mid \{ v_1 \mid v_2 \} \mid \text{rec } f \ x = t$$

$$t ::= v \mid \text{let } x = t_1 \text{ in } t_2 \mid v \{ v_2 \mid v \} \mid \text{if } v \text{ then } t_1 \text{ else } t_2 \mid v \{ v_1 : v_2 \mid \text{ref}(v)$$

Here, $x$ ranges over variables and constants $c$ over constant symbols, each of which has an associated interpretation $[c] \in V$; these include numerals $n$ with $[n] = \text{int}(n)$, booleans, arithmetic operations, test functions to tell whether a value is an integer, a function, a pair, or a reference, equality test for $R$-values, etc. $\text{rec } f \ x = t$ defines a recursive function with body $e$ and recursive calls made via $f$; we use $\lambda x. t$ as syntactic sugar in the case when $f \notin \text{fv}(t)$. Finally, $\text{let}(v)$ returns the contents of location $v$, $\text{ref}(v)$ allocates a new location initialized with $v$. The meta-theory is simplified by using “let-normal form”, in which the only elimination for computations is let, though we sometimes nest computations as shorthand for let-expanded versions in examples.

The untyped denotational semantics of values $[v] \in V \to V$ and terms $[t] \in V \to \mathcal{C}$ is defined in usual way by inductive clauses, which we omit here.

**Types** Types are given by the grammar

$$\tau ::= \text{unit} \mid \text{int} \mid \text{bool} \mid A \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

where $A$ ranges over user-specified abstract types. They will typically include region-indexed reference types such as $\text{intref}$, and also types like lists, sets, and even objects, again possibly refined by regions. The metavariable $e$ represents an effect, that is a subset of some fixed set of elementary effects about which we say more later. The core typing rules for values and computations are shown in Figure 1. The side condition on rule $\text{Dead}$ requires us to ascertain that $t_1$ terminates for all inputs and heaps satisfying the contracts specified in $\Gamma, x$.

We assume an ambient set of $\text{typing axioms}$ each having the form $(\nu, \tau)$ for $\nu$ a value in the metalanguage and $\tau$ a type meaning that $\nu$ is claimed to be of type $\tau$ and that this will be proved “manually” using the semantics rather than using the typing rules. We also assume an ambient set of $\text{equality axioms}$ each having the form $(\nu, \nu', \tau)$ and asserting semantic equality of $\nu$ and $\nu'$ at type $\tau$, again to be justified “manually”. We assume that whenever $(\nu, \nu', \tau)$ is an axiom so are $(\nu, \tau)$ and $(\nu', \tau)$.

The equational theory can be classified in three different categories: basic, congruence and effect-dependent. An extract of the basic and congruence equations are depicted in Figure 2. The theory also includes all the usual beta and eta laws and commuting conversions for conditionals as well as for $\text{let}$. We will give a semantic interpretation of typed equality judgments which is sound for observational equivalence.

As with typings, further equations involving effectful computations may be justified semantically in a particular model and added to the theory. In particular, we can justify the soundness of four effect-based equations, namely, dead, duplicated and commuting computation, and pure lambda hoist also shown in Figure 2. The core theory then allows one to deduce new semantic equalities from already proven ones.

4. Abstract Locations

We now define the concept of an abstract location which generalizes physical locations in that it models a portion of the store that can be read from and updated. Such portion may comprise a fixed set of physical locations or a varying such set (as in the case of a linked list with some given root). It may also reside in just a part of a physical location, e.g., comprise the two low order bits of an integer value stored in a physical location. Furthermore, the equality on such abstract location may be coarser than physical equality, e.g., two linked lists might be considered equal when they hold the same set of elements, and there may be an invariant, e.g., the linked list should contain integer entries and be neither circular nor aliased with other parts of the heap. This then prompts us to model an abstract location as a partial equivalence relation (PER) on heaps together with two more components that describe how modifications of the abstract location interact with the heap as a whole. Thus, next to a PER, an abstract location also contains a reflexive transitive relation (“guarantee”) modelling its evolution by way of modifying actions.

We will define a notion of when abstract locations are independent, generalizing the traditional notion of separation. However, there is some extra complexity due to the fact that whatever custom notions of equality and separation we introduce, we would like them all to interact well with the underlying built-in allocator. A
Table 1. Core rules for effect typing

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash t : \tau$</td>
<td>Basic Equations (Extract)</td>
</tr>
<tr>
<td>$\Gamma \vdash f : \tau_1 \rightarrow \tau_2, \Sigma \vdash e : \tau_2$</td>
<td>$\Gamma \vdash e_1 \Sigma \vdash e_2 : \tau_2 \rightarrow \tau_3$</td>
</tr>
</tbody>
</table>

**Figure 1.** Core rules for effect typing

**Figure 2.** Basic, congruence, and effect based equational theory

$$\pi \cdot 1 \equiv \pi(l) \quad \pi \cdot (a_1, a_2) = (\pi \cdot a_1, \pi \cdot a_2) \quad \text{dom}(\pi \cdot h) = \pi \cdot \text{dom}(h)$$

**Figure 3.** Action of permutations

Freshly allocated concrete cell should be independent of any existing abstract locations, for example, and abstract locations should not be sensitive to the behaviour of the underlying allocator. Were we working in a lower-level language (in which the allocator is just another piece of code to be verified), we would hope that concrete locations could be treated as a special case of abstract ones, but this seems to require a notion of when one notion of location is an abstraction of another (Jensen and Birkedal have taken some steps along such a path [25]). Instead, we here bake in compatibility with the underlying allocator using nominal ideas.

Following Gabbay and Pitts [21], we write $\text{perm}(L)$ for the group of permutations of $L$ and define an action $\pi \cdot -$ of permutations $\pi \in \text{perm}(L)$ for values, stores, sets of locations, and relations on heaps. Table 3 contains the most important cases: Note that permutations act differently on sets of locations and on sets of relations on heaps. In the sequel we call "object" anything that the permutations act on, i.e., locations, values, stores, etc. For example, a list of heaps in is in the relation $\pi \cdot R$ iff the list of heaps obtained by undoing the permutation ($\pi^{-1} \cdot h$) is in the relation $R$. We define the set of fixpoints of a permutation $\pi$ as $\text{fix}(\pi) = \{ l \mid \pi(l) = l \}$. For an object $x$, a support of $x$ is a set of locations $L$ such that $\text{fix}(\pi) \supseteq L \Rightarrow \pi \cdot x = x$. Gabbay and Pitts show [21] that if an object has a finite support then it has a unique (a least) support, which we write $\text{supp}(x)$.

It is clear that stores, values, and finite sets of locations always have a finite support. A store relation having a finite support is called finitely supported. For example, the predicate $P$ on stores asserting that there is a linked list starting at location $X$ has support $\{X\}$. Note that although a linked list in $h$ witnessing that $h \in P$ may have a large "footprint", the support still is just $\{X\}$ because the internal nodes of such a linked list will be consistently renamed by the permutation action.

**Definition 4.1** (Abstract Location). An abstract location $l$ (on the chosen set $\mathbb{E}$) consists of the following data:

- a non-empty, finitely supported, partial equivalence relation (PER) $\lhd$ on $\mathbb{E}$ modelling the "semantic equality" on the bits of the store that $l$ uses (a "rely-condition");
- a finitely supported transitive relation $\equiv$ modelling what bits of the store "writes to $l$" leave intact (a "guarantee condition").
That is, if \((h, h_1) \in \Gamma\) then \(h_1\) might arise by writing to \(l\) in \(h\) and all possible writes are specified by \(\Gamma\); subject to the conditions, where \(\pi\) is a permutation and \(h : \Gamma\) is shorthand for \((h, h_1) \in \Gamma\).

1. If \(\text{fix}(\pi) \not\supset \text{supp}(\pi)\) then \(h : \Gamma \Longrightarrow (h, \pi \cdot h) \in \Gamma\).
2. If \(h : \Gamma\) then \((h, h_1) \in \Gamma\).
3. If \((h, h_1) \in \Gamma\) then \(h : \Gamma\) and \(h_1 : \Gamma\).
4. If \(\Gamma, \Gamma_1 \subseteq \Gamma, \pi_1\), that is, if \((h, h_1) \in \Gamma\) and \((h, h_1) \in \Gamma_1\), then there exists \(h'\) such that \((h', \pi_1) \cdot h_1 \in \Gamma_1\) and \((h_1, h') \in \Gamma\).
5. If \(h : \Gamma\) and \(h_1 \supset h\) then \((h_1, h) \in \Gamma\) thus \(\Gamma\) “looks” no further than the currently allocated portion of the heap.
6. If \(h : \Gamma\) and \((h, h) \in \Gamma\), then \(\exists h_1, (h_1, h) \in \Gamma\) and \(\exists h' = h_1 \cup h'\).

Condition 1 asserts that semantic equality is closed under relocation. It rules out relations that stipulate equality of internal pointers (not to fixed ones which would enlarge the support, but between \(h\) and \(h'\)). Conditions 5 and 6 ensure that the relies and guarantees interact well with future extensions of the heap, preserving and being independent of fresh locations.

We write \(h \leftarrow h'\) to denote \((h, h') \in \Gamma\), \(h \rightarrow h_1\) to denote \((h, h_1) \in \Gamma\). We also use the notation \(\text{ supp}(\pi) = \text{ supp}(\pi) \cup \text{ supp}(\pi')\).

If \(\pi\) is a permutation we define \(\pi \cdot l = (\pi \cdot l_1, \pi \cdot l_2, \pi \cdot l_3)\); it is easy to see that \(\pi \cdot l\) is an abstract location and \(\text{ fix}(\pi) \supset \text{ supp}(\pi)\) \(\Rightarrow\) \(\pi \cdot l = l\).

Furthermore, for any \(\pi\) and \(\pi'\) such that \(\pi \supset \text{ supp}(\pi)\), one has \(\pi \cdot l = \pi \cdot \pi'\). This makes the following definition possible:

**Definition 4.2** (relocation of abstract locations). Let \(l\) be an abstract location and \(u \cdot l \supset \text{ supp}(\pi) \rightarrow \text{ L}\) be injective. Then \(u \cdot l\) is defined as \(\pi \cdot l\) for some \(\pi\) extending \(u\). We have \(\text{ supp}(u \cdot l) = u \cdot \text{ supp}(l)\).

For an example, consider the following abstract location \(\text{ set}_X\), with support \(X\), implementing the set example introduced above:

\[
\begin{align*}
\text{ set}_X & = \{(h, h') | \text{ If } h \text{ and } h' \text{ contain a linked list starting from } X \\
& \text{ and these lists contain the same set of integers.}\}
\end{align*}
\]

where \(F(h, X)\) is the set of the locations in \(h\) reached by the linked list starting from \(X\). The rely set, \(\text{ set}_X\), specifies that two heaps are equivalent when linked lists denote the same set of integers. The guarantee, \(\text{ set}_X\), specifies that a write to this abstract location may not change concrete locations not in the linked list.

Note, in particular, how this example validates condition Def 4.1(1): We can relocate one of the two linked lists without compromising relatedness in \(\text{ set}_X\) provided we leave the entry point \(X\) fixed. If one tried to modify our abstract location to relate linked lists with the exact same footprint, then property 1 would fail.

**Definition 4.3** (Separated Heap). Let \(l_1, l_2, \ldots, l_n\), be abstract locations. A heap \(h\) is associated with respect to \(l_1, l_2, \ldots, l_n\), written \(\text{ sep}(h, \{l_1, l_2, \ldots, l_n\})\), if for all \(1 \leq i \leq n, h : \Gamma\) and for all \(j \neq i\) and for any heap \(h_1\) such that \(h \leftrightarrow h_1\), we have \(h_1 \sim h_1\), and, coinductively, \(\text{ sep}(h, \{l_1, l_2, \ldots, l_n\})\).

**Definition 4.4** (Independence). Abstract locations \(l_1, l_2, \ldots, l_n\) are independent, written \(\perp\{l_1, l_2, \ldots, l_n\}\), if \(h_1 : \{l_1, h_1 : \{l_1, \ldots, H, l_n\} \text{ then these exist } h, \text{ where supp}(h) = \{l_1, \ldots, \text{ and } h \sim h_1\}\) and \(h \sim h_1\) for \(1 \leq i \leq n\).

The definition of independence implies that quotienting by the intersection of the PERs yields a product of the individual quotients. As the separated heap defined above is unique up to \(\sim\) with respect to \(l_1, \ldots, l_n\), we use the notation \(h_1 @ h_2 @ \cdots @ h_n\) to denote a canonical separated heap.

Returning to the setfactory example, consider two sets \(\text{ set}_X\) and \(\text{ set}_Y\), with \(X\) and \(Y\) the concrete locations where the linked lists implementing, respectively, \(\text{ set}_X\) and \(\text{ set}_Y\) start. If \(X\) and \(Y\) are different, we use coinduction to show \(\text{ set}_X\) and \(\text{ set}_Y\) are independent, using the following set as the witnessing co-inductive hypothesis for the separated heaps requirement:

\[
P_{\text{ set}} \sim \{h | \exists h_1, h_2, \text{ if } h = h_1 @ h_2 \land (X \rightarrow 0) \text{ int}(h_1) \land (Y \rightarrow 0) \text{ int}(h_2)\}.
\]

**Remark 4.12**. Motivated by the axioms of separation algebras \([13]\), one might conjecture that \(\oplus\) is cancellative, i.e., \(l_1 @ l_2 = l_3 \oplus l_2\) implies \(l_1 = l_3\). However, for the above definitions, this does not generally hold.

### 4.1 Examples of Abstract Locations

Besides abstraction locations for sets of integers used above, we illustrate other instances of abstract locations:

**Single Integer** For our simplest example, consider an abstract location \(l_x\), parametric with respect to \(X\) as follows:

\[
\begin{align*}
\text{int}_x & = \{h | h(X) = n \land h'(X) = n' \Rightarrow \text{int}(n) = \text{int}(n')\} \\
\text{int}_y & = \{h | h(X) = n \land \text{int}_x(h) = \text{int}_y(h) \land \text{int}_x(h) \land \text{int}_y(h) = (h(0) = h(1))\}
\end{align*}
\]

Two heaps are in its rely relation if the values stored in \(X\) are the same; and its guarantee is to leave all other concrete locations alone.

It is also the case that \(\text{ int}_x\) and \(\text{ int}_y\) are independent, i.e., \(\text{ int}_x \perp \text{ int}_y\). This is witnessed by the following co-inductive hypothesis to show the separateness condition of heaps with respect to these locations, similar to the one used before for the showing of independence of the \(\text{ set}_X\) and \(\text{ set}_Y\):

\[
P_{\text{ int}} \sim \{h | \exists h_1, h_2, \text{ if } h = h_1 @ h_2 \land (X \rightarrow 0) \text{ int}_x(h_1) \land (Y \rightarrow 0) \text{ int}_y(h_2)\}.
\]

**Overlapping references** Recall the overlapping references example introduced earlier. Let \(X\) be the concrete location encoding a
A pair of morphisms $\lambda_3\beta_i^0 = \{(h, h') \mid h(X) = 0 \Rightarrow \text{initial}(h', X) \text{ and } h'(X) = 0 \Rightarrow \text{initial}(h, X) \text{ and } h(X) = (a_i, a_\lambda) \wedge h'(X) = (a'_i, a'_\lambda) \Rightarrow h'(a_i) = h'(a'_i)\}$

The rely specifies that two heaps are equivalent when either $X$ is in the initial, uninitialized, state in both of them, as specified by the predicate $\text{initial}$, or $X$ in both heaps points to a pair of references, the i-th of which also contain the same values. The guarantee condition specifies that $X$ in an updated heap is 0 only if it was so before, and moreover all locations not used in the rely are left unchanged.

5. Setoids

Abstract locations capture the invariants that we will use to build our model. The ‘big picture’ is that types are modelled as functors from a category of worlds into a category of setoids, with terms being interpreted as natural transformations. Worlds will comprise independent abstract locations, whilst setoids are predomains augmented with a proof-relevant notion of equality. In this section, we define the structure of setoids more precisely. Section 6 then explains the structure of the category of worlds and setoid-valued functors.

While we assume that the reader is familiar with some notions of category theory, such as functor, natural transformation, and cartesian closure [31], we review some basic concepts and notation here. We compose morphisms in the usual applicative order, thus, if $f : A \to B$ and $g : B \to C$ then $gf : A \to C$. A morphism $u$ in a category $C$ is a monomorphism if $ux = ux'$ implies $x = x'$ for all morphisms $x, x'$. A commuting square $ux = x'ux'$ of morphisms is a pullback if whenever $vx = v'x'$ there is unique $t$ such that $v = ut$ and $v' = ut'$. A pair of morphisms $u, u'$ with common domain is a span, a pair of morphisms $x, x'$ with common codomain is a co-span. A category has pullbacks if every co-span can be completed to a pullback square.

We define the category of setoids as the exact completion of the category of predomains, see [10, 14]. We recall here the elementary description using the language of dependent types given in an earlier paper et al. [4].

Definition 5.1. A setoid $A$ consists of a predomain $\{A\}$ and for any two $x, y \in \{A\}$ a set $\{x = y\}$ of “proofs” (that $x$ and $y$ are equal). The set of triples $\{(x, y, p) \mid p \in \{A, x, y\}\}$ must itself be a predomain and the first and second projections must be continuous. Furthermore, there are continuous functions $r_A : \Pi x : \{A, x\} \times \{A, x\} \to \{A, x, y\}$ and $t_A : \Pi x, y, z : \{A, x, y\} \times \{A, y, z\} \to \{A, x, z\}$, witnessing reflexivity, symmetry and transitivity; note that no equations between these are imposed.

In the above definition, continuity of a dependent function such as $\lambda(x, y)\Rightarrow f(x, y)$ means the following: If $(x_0, y_0)$ and $(x_1, y_1)$ are ascending chains in $A$ with suprema $x, y$, then $f(x_0, y_0)$ is an ascending chain of proofs (by monotonicity of $\lambda(-, -)$) and its supremum is $f(x, y)$.

If $p : A(x, y)$ we may write $p : x = y$ or simply $x = y$. We also omit $[-]$ wherever appropriate.

Setoids based on types rather than predomains have been used to provide extensional equality in intuitionistic type theory [2].

Definition 5.2. A morphism from setoid $A$ to setoid $B$ is an equivalence class of pairs $f = (f_0, f_1)$ of continuous functions where $f_0 : \|A\| \to \|B\|$ and $f_1 : \Pi x, y, z : \|\{A, x, y\}\| \to \|\{B, f_0(x), f_0(y)\}\|$. Two such pairs $f, g : A \to B$ are identified if there exists a continuous function $\mu : Pa : \|A\| \to \|B(f(a), g(a))\|$. Morphisms are composed in the obvious way to form the category of setoids. In the sequel we tend to omit $[-]$ and 0, 1-subscripts where appropriate, thus writing $x(a)$ for $x_0(a)$ or $a \in A$ or even $a : A$ for $a \in \{A\}$ etc.

We remark that one could also choose not to identify $\sim$-equal morphisms. We decided to make the identification because standard category-theoretic concepts like cartesian closure, pullbacks, etc. then apply directly, rather than only holding up to $\sim$.

The following is folklore, see also [10].

Proposition 5.3. The category of setoids is cartesian-closed. The cartesian product $A \times B$ of setoids $A, B$ has components $\|A \times B\| = \|A\| \times \|B\|$ and $\|\{A, (a, b)\}\| = \|\{A, a\} \times \{B, b\}\|$. The ordering is given component-wise.

The underlying predicate of the function space of $A \to B$ of setoids $A, B$ is given by the set of representatives $(f_0, f_1)$ of morphisms from $A$ to ordered component- and point-wise. Given two such elements $f = (f_0, f_1)$ and $g = (g_0, g_1)$ a proof $\mu : f \sim g$ in $\|A\| \Rightarrow \|B\|$ is a continuous function establishing equality of $f$ and $g$ qua morphisms from $A$ to $B$ as in Definition 5.2.

Proof The verification of the cartesian product is straightforward. The application morphism $ap$ from $A \Rightarrow B \times A$ to $B$ has components given by $ap_0((f_0, f_1), a) = f_0(a)$ and $ap_1((\mu, p)) = f_1(p); \mu(a')$ where $\mu : f \sim g$ and $p : a = a'$. Note that we have $f_1(p) ; f_0(a) \sim f_0(a') \mu(a') \sim f_0(a') \mu'(a'') \sim f_0(a'')$. Conversely, if $h : C \times A \to B$ is a morphism represented by $(h_0, h_1)$ we define $\lambda(h) : C \Rightarrow A \to B$ by $(\lambda(h))(c) = (f_0, f_1)$ where $f_0(a) = h_0(c, a)$ and $f_1(p) = h_1(r_0(c), p)$. We also put $(\lambda(h))(p ; c \sim c') = h_1(p, r_1(a)).$ (Here $h_0, h_1$ is another representative of $h$ witnessed by a proof $\mu$ and $\lambda(c), \lambda(d)$ witnesses equality of the so constructed abstraction. The remaining verifications are direct.

Proposition 5.4. The category of setoids has pullbacks.

Proof To construct a pullback of $x : A \to B$ and $x' : A' \to B$ pick representatives $(x_0, x_1)$ and $(x'_0, x'_1)$ and take the low
point of the pullback $U$ to be given by $|U| = \{(a, a', p) \mid a \in A, a' \in A, p \in B(x(a), x'(a'))\}$ and then $U(a, a', p) = (a, a') = A(a, a') \times A(a, a')$. The ordering is component wise. The morphism $u : U \to A$ is defined by $u(a, a', q) = a$ and $u(p, p') = p$. The remaining definitions and verifications are straightforward. Note, however, that proving the uniqueness of fill-ins to establish the pullback property we make use of the fact that pointwise equal morphisms are identified.

Definition 5.5. A setoid $D$ has a least element if $|D|$ has a least element $\bot$ and there is also a least proof $\bot \in D(\bot, \bot)$.

Proposition 5.6. For each $D$ with least element there is a morphism $Y_D : (D \Rightarrow D) \to D$ such that the following hold:

- $Y_D = ap(id_{D(\bot, \bot)}, Y_D)\text{ (so }Y_D(f)\text{ is a fixpoint of }f)$.
- Suppose that both $D, D'$ have least elements and $u : D \to D'$ satisfy $u(\bot) = \bot$. Let $D \Rightarrow D' \rightsquigarrow \bot \Rightarrow \bot \Rightarrow D'$ be the pullback of $u(\bot) : D \Rightarrow D \Rightarrow D'$ and $u(\bot) : D' \Rightarrow D \Rightarrow D'$. So, intuitively, $X$ contains pairs $(f, f')$ with $f : D \Rightarrow D'$, $f' : D' \Rightarrow D$ and $uf = 1u$. Then $Y_{D \Rightarrow D'} = Y_{D \Rightarrow D'}$. Intuitively, this means that if $\mu f = \mu f'$ and $u(\bot) = 1$, then $u(Y_{D}(f)) = Y_{D}(f')$.

Proof. Given $f \in D \Rightarrow D$ construct the least fixpoint of $f$ as sup $f, f(\bot)$. The passage from $f$ to the least fixpoint being continuous this defines the first part of $Y_D$, i.e. $Y_D(f) = \sup f, f(\bot)$. As for the proof component assume that $p : f \sim f'$. By repeatedly invoking $p$ we then get a family of proofs $p_i : f, f(\bot) \sim f, f(\bot)$). Using the fact that $r(\bot)$ is the least proof we can then show by induction on $i$ that this family of proofs is increasing and its supremum satisfies $\sup p_i : Y_D(f) \sim Y_D(f')$. Thus, we put $(Y_D)_i(p) = \sup p_i$, which is again continuous.

It is obvious from the construction as a least fixpoint the required fixed point equation is indeed satisfied. As for the second one pick $u : D \Rightarrow D'$ and assume $f : D \Rightarrow D$ and $f' : D' \Rightarrow D'$ and a proof $p : uf \sim f'u$. Now, by repeatedly invoking $p$ we get an increasing family of proofs $p_i : uf, f(\bot) \sim f', f(\bot)$ whose supremum witnesses $uf(Y_D(f)) \sim Y_{D'}(f')$. This establishes the claim.

6. Worlds and Setoid-Valued Functors

We now organize abstract locations into a category that will index the meanings of types and terms in a functor category semantics [33, 39]. Our category is morally similar to that of finite sets and injections, which has been used in other work [6, 39]. The difference is that objects are now sets of mutually independent abstract locations and morphisms are slightly more complicated in that they allow relocation of abstract locations. Furthermore, locations are grouped into regions allowing one to approximate them statically.

We begin with some more notational preliminaries. Write $x, x', y, y'$ or $w, w'$ or $w, w'$ (when $w'' = \operatorname{dom}(w')$) for a pullback square with morphisms $x, x', y, y'$. We call the common codomain of $x$ and $x'$ the apex of the pullback written $\bar{w}$, while the common domain of $u, u'$, the low point of the square, is written $w$. A pair of morphisms $u, u'$ with common domain is a span, a pair of morphisms $x, x'$ with common codomain is a co-span. A category has pullbacks if every co-span can be completed to a pullback square. A pullback square $w, w', w$ with apex $\bar{w}$ is minimal if whenever there is another pullback $w, w', w'$ over the same span and with apex $\bar{w}$, then there is a unique morphism $t : \bar{w} \to \bar{w}$ such that $x_1 = tx$ and $x_1' = tx'$.

Lemma 6.1. If a category $C$ has pullbacks and $w, w', w$ with apex $\bar{w}$ is a minimal pullback then the morphisms $x$ and $x'$ are jointly epic, that is, if $fx = gx$ and $f'x = g'x'$ then $f = g$.

For illustration purposes, we notice that the category of finite sets and injections has pullbacks and in it every span can be completed to a minimal pullback. Indeed, if $u : U \to A$ and $u' : U \to A'$ then the apex of the minimal pullback can be chosen as $A \setminus \operatorname{Im}(u) + A'$ with + denoting disjoint union.

Definition 6.2 (Category of worlds). Assume given an infinite set $\text{Regs}$ of region names. The category of worlds $\text{W}$ is defined as follows:

- An object (written $w$ and called a world) comprises a finite set of mutually independent abstract locations written $\text{dom}(w)$ and a function tagging each abstract location in $w$ with a region from $\text{Regs}$; we write $w(t)$ for the set of abstract locations in $w$ tagged with $t$. We define $\operatorname{supp}(w) = \bigcup_{t \in \text{dom}(w)} \operatorname{supp}(t)$.
- Let $w$ and $w'$ be two worlds. A morphism from $w$ to $w'$ is given by an injective function $u : \operatorname{supp}(w) \to \operatorname{supp}(w')$ such that for each $l \in \text{dom}(w)$ we have $u \cdot l \in \text{dom}(w')$ (cf. Def. 4.2) and moreover, if $l \in \text{w}(t)$ then $u \cdot l \in w'(t)$.
- We write $\text{I}(w, w')$ for the set of morphisms $u : w \to w'$ that are set-theoretic inclusions. Note that $u \cdot l = l$ in this case.
- The empty world is denoted $\emptyset$, i.e. $\text{dom}(\emptyset) = \emptyset$.

For example, a morphism between $\{\text{int}_1\} \to \{\text{int}_2, \text{set}_2\}$ (region tags omitted) is given by the injective function that simply maps the concrete location $X$ to the concrete location $Y$.

The abstract locations grouped together in one region represent all the stateful components, be they physical locations or more complex updatable data structures, that belong to that region. Static effect annotations amalgamate possible side effects on any of those components as effects on the region.

The notion of morphism presented here is the simplest possible one that allows us to justify interesting effect-dependent equivalences. There are more general notions of morphism that allow one to justify equivalences relying on representation independence, e.g. ones resulting from replacing $\text{int}_1$ by $\text{int}_2$ in a program.

It is also possible to define the category of worlds differently by combining all the abstract locations contained in a given region using the $\diamond$-operation. A world then becomes a function assigning to each region a single ‘large’ abstract location, possibly the trivial one. The presence of a morphism from one world to another then has to imply that the abstract locations attached to a region grow properly, i.e., go from 1 to an abstract region isomorphic to $1 \otimes 1'$, which can be elegantly captured by adopting Oles’s [33] idea of morphisms between store shapes, which has later been popularized as lenses in the context of bidirectional synchronization [20].

We reluctantly decided, for reasons of space and simplicity, against using these more general notions of worlds and morphisms here, but plan to discuss them in an extended version of the paper.

The following is proved just as in the case of finite sets and injections.

Proposition 6.3. In the category $\text{W}$ all morphisms are monomorphisms; $\text{W}$ has pullbacks and all spans in $\text{W}$ can be completed to minimal pullbacks.

Moreover, these pullbacks and minimal pullbacks can be chosen in such a way that if one of the two given maps is an inclusion then the one parallel to it is also an inclusion.

If $u : w \to w_1$ and $x : w_1 \to w_2$ then we can find $w'_1$ and $u' : w \to w'_1$ and $x' : w'_1 \to w_2$ so that $w'_1, x', w'_1, x' \to w_2$ is a minimal pullback and if one of $u, x$ is an inclusion then the opposite morphism can be chosen as an inclusion as well.

The world $w'_1$ in the last clause comprises the locations in $w$ and those in $w_2$ that are not in $w_1$. Intuitively, if $w = A, w_1 = A + B, w_2 = A + B + C$ then $w'_1 = A + C$.

6.1 Setoid-valued functors

A functor $A$ from a category of worlds $\text{W}$ to the category of setoids comprises as usual for each $w \in \text{W}$ a setoid $A w$ and for each $u : w \to w'$ a morphism of setoids $A u : A w \to A w'$ preserving identities and composition. If $u : w \to w'$ and $a \in A w$ we may
write \( u.a \) or even \( a.a \) for \( Au(a) \) and likewise for proofs in \( Aw \). Note that \((uv)v = u.(v.a)\).

**Definition 6.4.** We call a functor \( A \) pullback-preserving (p.p.f.) if for every pullback square \( \square \) with apex \( \bar{w} \) and low point \( w \) the diagram below is a pullback in the category of setoids.

\[ \begin{array}{ccc}
Aw & \to & A w \\
\downarrow & & \downarrow \\
Aw & \to & A w
\end{array} \]

Given our characterization of pullbacks in the category of setoids (Prop. 5.4), this condition is equivalent to the existence of a continuous function of type

\[ \Pi a \in A w. \Pi a' \in A w. A w(x.a, x'.a') \to \Sigma a \in A w. A w(u.a, a) \times A w(u'.a', a') \]

which amounts to saying that if two values \( a \) and \( a' \) are equal in a common world \( w \), then this can only be the case because there is a value in the "intersection world" \( w' \) from which both \( a \) and \( a' \) arise. The witness \( \bar{a} \) is unique up to \(-\). More importantly, even, two values cannot suddenly become equal just because we move to another world. Even though pullback preservation is not formally needed in the proofs of any of the subsequent results it simplifies the intuition considerably.

**Lemma 6.5.** If \( A \) is a p.p.f. and \( u : w \to w' \) and \( a', a' \in A w \) then there is a continuous function \( A w(u.a, u.a') \to A w(a', a') \). (There is also a continuous function in the other direction just by functoriality.)

Note that the ordering on worlds and world morphisms is discrete so that continuity only refers to the \( A w(u.a, u.a') \) argument.

**Definition 6.6 (Morphism of functors).** If \( A, B \) are p.p.f.s, a morphism from \( A \) to \( B \) is a natural transformation from \( A \) to \( B \). Thus, concretely, a morphism is an equivalence class of pairs \( e = (e_0, e_1) \) of continuous functions where \( e_0 : \Pi w. A w \to B w \) and \( e_1 : \Pi w. A w. \Pi x. A w(x.a, x.a') \to B w(x.a, x.a') \).

Two such morphisms \( e, e' \) are equal if there exists a continuous function \( u : \Pi w. A w. B w(e(a), e(a')) \).

These morphisms compose in the obvious way and so the p.p.f.s and morphisms between them form a category.

**Theorem 6.7.** The category of p.p.f.s is cartesian-closed and has pullbacks. If \( A, B \) are p.p.f. then \((A \times B)w = A w \times B w\) and \( \Pi (A \Rightarrow B w) \) contains pairs \((f_0, f_1)\) as follows: \((f_0(u), f_1(u))\) for each \( w \), and \( u : w \to w' \) then \( f_1(u, v) \in (A w' \Rightarrow B w)(f_0(u), f_1(u), (w', v)) \).

These proofs are the obvious composition morphisms.

A proof in \((A \Rightarrow B w)(f_0(u), f_1(u), (w', v))\) is a function \( g(u) \in (A w \Rightarrow B w)(f_0(u), f_1(u), (w', v)) \). The order on objects and proofs is pointwise as usual.

Moreover, if \( D \) is a p.p.f. with the property that \( D w \) has at least element \( \bot \) for each \( w \) and whenever \( u : w \to w' \), then \( u.\bot \to \bot \), there is a morphism \( Y_D : (D \Rightarrow D) \to D \) satisfying the same properties as those asserted in Prop. 5.6.

**Proof** This is a special case of the construction of a functor category with respect to an internal category (here the category of worlds viewed as discrete setoids, thus internal to the category of setoids). One mimics the standard constructions for set-valued functors taking care to keep representatives of morphisms distinct as we did in the construction of the function space of setoids.

The existence of the fixpoint operator does not seem to be known but follows by applying the same method. The rather fine-grained specification of the fixpoint operator on the level of setoids is needed here so that we can assure naturality of \( Y_D \) and \( Y_D(f) \).

**Heaps as a setoid.** We will now equip heaps with a world-dependent setoid structure singling out the separated ones (for a particular world) and setting equality to be induced by the ambient rely relations.

**Definition 6.8.** If \( w \) is a world then the setoid \( \mathcal{S} w \) is defined by \( \mathcal{S} w = \{ h \in H \mid \forall l : w, h : l \perp \wedge \text{sep}(h, \text{dom}(w)) \} \) and \( \mathcal{S} w(h, h') = (\star) \iff \forall l : w, h \perp h' \wedge \mathcal{S} w(h, h') = \emptyset \) otherwise, where \( \text{sep}(h, \text{dom}(w)) \) is the separated heap with respect to the locations in \( \text{dom}(w) \) (see Definition 4.3).

If \( u : w \to w' \) and \( h \in \mathcal{S} w \), then we define \( h.u \in \mathcal{S} w \) to be \( \pi^{-1} \cdot h \) for some \( \pi \) extending \( u \). This is well-defined up to \(-\).

In this way, \( \mathcal{S} \) becomes a contravariant functor from the category of worlds to the category of setoids.

**Lemma 6.9 (Pasting Lemma).** Let \( w \Rightarrow w' \) be a minimal pullback with apex \( w \). Let \( h \in \mathcal{S} w \) and \( h' \in \mathcal{S} w' \) such that \( h \perp h' \). Then there exists a heap \( h_1 \in \mathcal{S} w \), unique up to \(-\)

**Corollary 6.10.** Let \( u : w \to w' \) be a morphism and \( h \in \mathcal{S} w \) and \( h' \in \mathcal{S} w' \). There exists \( h_1 \) such that \( h_1 \perp h \) and \( h_1.u \perp h' \). Let \( w_1, w_1' \) be a minimal pullback with low point \( \emptyset \), the empty world.

Intuitively, \( h_1 \) is obtained by overwriting the \( w \)-part of \( h' \) according to \( h \).

**Heap Relations.**

**Definition 6.11.** A relation \( R \) on \( \mathcal{S} \) consists of a subset \( R w \subseteq \mathcal{S} w \times \mathcal{S} w \) for each world, such that if \((h, h') \in R w \) then \((h.u, h'.u) \in R_{w_0} \) and if \( p : h \Rightarrow h' \) then \((h_1, h'_1) \in R_w \). Given a minimal pullback \( w \Rightarrow w' \) and \( h, h' \in \mathcal{S} w \) such that \((h.x, h'.x) \in R_w \) then \((h', h') \in R_w \).

The elementary effects track reading, writing, and allocating at the level of regions and as follows: \( \text{wr}_r \) (writing within region \( r \), \( \text{rd}_r \) (reading from within region \( r \), and \( \text{al}_r \) (allocating within region \( r \)). Each elementary effect \( e \) is associated with a set \( R(e) \) of relations on \( \mathcal{S} \) as follows:

\[ R \in R(\text{rd}_r) \iff (h, h') \in R w \Rightarrow \forall l : w(r), h \perp h' \]

\[ R \in R(\text{wr}_r) \iff (h, h') \in R w \Rightarrow \forall l : w(r), h \perp h' \]

\[ R \in R(\text{al}_r) \iff (h, h') \in R w \Rightarrow \forall l : w(r), h \perp h' \]

Thus, \((R, \text{rd}_r), (R, \text{wr}_r), (R, \text{al}_r) \) is the set of relations for which \( R \)-related heaps contain "equal" (in the sense of \( \perp \)) values for all abstract locations in region \( r \), a relation \( R \in R(\text{wr}_r) \) is oblivious to (preserved by) "equal" writes to any abstract location in \( r \), and a relation \( R \in R(\text{al}_r) \) is oblivious to extensions of the current world provided that they only add abstract locations in region \( r \), that the initial contents of these newly allocated locations are "equal" in the sense of \( \perp \), and that nothing else is changed.
Finally, $R(e)$ for a set of effects $e$ is defined as follows:

$$R(e) = \bigcap_{e \in e} R(e).$$

As in our earlier work [7], the key idea is that computations with effect $e$ will preserve all the heap relations that are preserved by all the operations allowed by $e$. The larger the set of operations, the fewer relations need be preserved.

**Definition 6.12 (Monad).** Let $A$ be a p.p.f. and $e$ an effect. A p.p.f. $T_A$ is defined as follows:

- **(Objects)** Elements of $(T_A)_w$ are pairs $(c_0, c_1)$ of partial functions where
  
  $$c_0 : \mathbb{W} \to \Sigma w. I(w, w_1) \times \mathbb{W} w \times A w_1$$

  and $c_1$ is as follows. If $R \in R(e)$ and $(h, h') \in Rw$ then $c_1(R, h, h')$ either is undefined and $c_0(h)$ and $c_0(h')$ are both undefined or else $c_1(R, h, h')$ is defined and then $c_0(h)$ and $c_0(h')$ are both defined, say $c_0(h) = (w_1, u, h_1, a)$ and $c_0(h') = (w'_1, u', h'_1, a')$. In this case, $c_1(R, h, h')$ returns a pair $(\langle \diamond \phi_1^r, p \rangle, \langle \diamond \phi_2^r, q \rangle)$ where $w_1^r \phi_1^r, w_1^r \phi_2^r$ such that $xu = xu'$. Furthermore, $p \in A \langle w_1 x a, x' a' \rangle$ and, finally, $(h_1, v, h'_1, v') \in Rw$ where $\mathbb{W}$ and $\mathbb{W}$ are locally and point of $\langle \diamond \phi^r \rangle$.

- **(Proofs)** Proofs only look at the $\langle - \rangle$-components. Let $c = (c_0, c_1) \in T_A w$ and $c' = (c_0', c_1') \in T_A w$, then a proof in $(T_A w)(c')$ is a partial (continuous) function $\mu$, such that for a given $h \in \mathbb{W}$:
  
  1. if $\mu(h)$ is undefined, then $c_0(h)$ and $c_0(h')$ are both undefined;
  2. if $\mu(h)$ is defined, then $c_0(h) = (w_1, u, h_1, a)$ and $c_0(h') = (w'_1, u', h'_1, a')$ are both defined. In this case, $\mu(h)$ returns a tuple $(\langle \diamond \phi_1^r, p \rangle, q)$ satisfying $xu = xu'$, $q \in A \langle w_1 x a, x' a' \rangle$ and $h_1, v, h'_1, v' \in \mathbb{W}$ with $\mathbb{W}$ and $\mathbb{W}$ apex and low point of $\langle \diamond \phi^r \rangle$.

- **(Order)** The order between computations is given point-wise: $(c_0, c_1) \leq (c_0', c_1')$ if and only if for any $h \in \mathbb{W}$ we have $c_0(h) \leq c_0'(h)$, which is defined as follows: for all $h \in \mathbb{W}$ if $c_0(h)$ is defined so is $c_0'(h)$. Moreover, if both $c_0(h)$ and $c_0(h')$ are defined as $c_0(h) = (w_1, u, h_1, a)$ and $c_0(h') = (w'_1, u', h'_1, a')$, then $w_1 = w'_1$, $u_1 = u'_1$, $h_1 = h'_1$ and $a_1 \leq a'_1$. Finally, if $c_0(R, h, h')$ and $c_0(R, h, h')$ are defined returning $(\langle \diamond \phi_1^r, p \rangle, q)$ and $(\langle \diamond \phi_2^r, q' \rangle, q'')$, then $\langle \diamond \phi_1^r, p \rangle \leq \langle \diamond \phi_2^r, q' \rangle$ where $\langle \diamond \phi^r \rangle$ and $q \leq q'$.\(^1\)

- **(Morphism)** Let $x : w \to w'$ be a morphism and $\mathbb{C} = (c_0, c_1) \in T_A w$. We are going to describe the first component of $(c_0, c_1) = T_A(x \circ c)$. Given $h \in \mathbb{W}$ write $c_0(h) = (w_1, u, h_1, a)$. This application be undefined so is $c_0(h)$. Now, using Prop. 6.3, complete $u, x$ to a minimal pullback square of $x \phi^r, x \phi^r$ with $u' : w \to w'$ where $w'$ is the apex of this pullback. Now, obtain $h' \in \mathbb{W}'$ from $h$ by overwriting with $h_1, u$ using Corollary 6.10. Then paste $h_1$ and $h'$ (note that $h_1, u \sim h', u$ by construction) using Lemma 6.9 to yield the desired heap $h'_1 \in \mathbb{W}'$. We then put $c_0(h) = (w'_1, u', h'_1, a')$.

We omit the remaining components and verifications.

The proof of the following is tedious but straightforward and will be given in an extended version of this paper.

**Theorem 6.13.** $T_e$ is the functor part of an effect-indexed strong monad [8, 43].

This theorem means in particular, that each $T_e$ is a monad on the category of p.p.f. i.e., for each $A$ there are natural transformation $\eta_A : A \to T_e A$ and $\mu_A : T_e T_e A \to T_e A$ subject to the usual laws and whenever $e_1 \subseteq e_2$ then there is a morphism $T_{e_1} A \to T_{e_2} A$ which interacts with the monad data in the expected sense.

\(^1\) Strictly speaking, $q \leq q'$ needs to be a tuple with the objects they are proving the equality of.

**Illustration** Assume an initial heap $h$, where $h(X) \to \{1, 2, 3\}$ and $h(Y) \to \{3, 2, 1\}$, that is, $h$ points to the lists $\{1, 2, 3\}$ and $\{3, 2, 1\}$ from concrete locations $X$ and $Y$, respectively. Moreover, let $c(h) = (w_1, u, h_1, (3, Y))$ and $c'(h) = (w'_1, u', h'_1, (3, Z))$, where $h_1(X) \to \{1, 2, 3\}; h(Y) \to \{3, 2, 1, 0\}$ and $h_1(Z) = \{4, 2\}$, while $h'(X) \to \{3, 2, 1, 0\}; h'(Z) \to \{3, 2, 1, 3, 2, 1, 0\}$ and $h'(W) = \{0, 0\}$. We can show that $c$ and $c'$ are equal when given $h$. The proof is illustrated by the diagram below, where an arrow labeled with $\subseteq$ denotes an inclusion and an arrow labeled with a mapping, e.g., $\subseteq \to Y$, denotes the corresponding renaming.

Here, it is easy to check that $h_1$ and $h_1'$ are equal when taken to the world $\mathbb{W}$, as the concrete locations $X$ and $Y$ and $Z$ are equal. Notice that the abstract locations $s_1 \subseteq w_1$ and $s_1 \subseteq w'_1$ are not taken into account. Moreover, the value $(\langle 3, Y \rangle)$ is also equivalent to $(\langle 3, Z \rangle)$ when taken to the world $\mathbb{W}$, as the morphism $w'_1 \to \mathbb{W}$ renames $Z$ to $Y$, while the morphism $w_1 \to \mathbb{W}$ is an inclusion.

We see here that the use of setoids rather than plain sets or predicates is crucial. The proofs in $T_A$ are clearly relevant as they explain in what way the abstract locations correspond to each other. Likewise, the elements of $|T_A|$ contain more information that the mere computation, i.e., the evolution of the heap. Namely, the world extension $w_1$ will typically contain new abstract locations with new contracts that can be arbitrarily complicated. Thus, it would not be possible to replace $T_A$ with a simple logical predicate on actual computations.

7. **Proof-relevant Logical Relations**

The structure identified so far allows us to interpret our effect type system and the equational theory in the category of p.p.f.

**Definition 7.1 (type interpretation).** A type interpretation consists of an assignment of a p.p.f. $[A]$ to each basic type $A$.

Given a type interpretation we can then assign a p.p.f. $[\tau]$ to each type the essential clause being $[\tau_1 \to \tau_2] = [\tau_1] \Rightarrow [\tau_2]$. We also interpret a typing context $\Gamma$ as the cartesian product over its bindings: $[\Gamma] = \prod_{\Gamma \text{red}} [\Gamma(\tau)]$.

**Lemma 7.2 (masking).** Let $\tau$ be a region. For world $w$ define $w \tau$ by removing all locations in $\tau$. If type $\tau$ does not contain $\tau$ then for each world $w$ the morphism $u.(\cdot) : [\tau] w \tau \Rightarrow [\tau] w$ where $u : w \tau \to w$ is the obvious inclusion is an isomorphism.

**Definition 7.3 (axiom interpretation).** An axiom interpretation consists of an element $x_{\tau_1, \tau_2} \in [\Gamma] \emptyset$ for each type axiom $(\tau, \tau)$ and of an element $p_{\tau, \tau'} \in [\Gamma] \emptyset(x_{\tau, \tau'}, x_{\tau', \tau})$ for each equality axiom $(\tau, \tau')$.

Given an axiom interpretation (and a type interpretation) and a well-typed term $\Gamma \vdash e : \tau \& e : [\Gamma] \Rightarrow T_{\tau}[e]$ using the fact that all the typing rules are reflected in the semantics: cartesian closure takes care of application and abstraction and
Lemma 7.4. Let $\Gamma \vdash e : \tau$ & $\epsilon$. Let $w \mathsf{w}_{\omega}^{\omega} \mathsf{w}$ be a pullback square, $\gamma \in \Gamma \mathsf{w}_{\omega}^{\omega}$ be a context, such that $\gamma : x : \chi \sim x : \chi$ and $C = (\Gamma \vdash e : \tau$ & $\epsilon \mathsf{w}(\gamma)) \in \mathcal{T}_{\omega_{\omega}}$ be computations, such that $\mu : x : \chi \sim x : \chi$. Let $h \in \mathsf{w}_{\omega}^{\omega}, h' \in \mathsf{w}$ which specifies that the heaps $h$ and $h'$ are equivalent on all the abstract locations $l$ in regions associated to read effects in $e$ and on the not write locations, respectively.

**Effects**

Given a set of effects $\epsilon$, we write $\mathsf{rds}(e) = \{r | | r \in \epsilon\}$, $\mathsf{wrs}(e) = \{w | w \in \epsilon\}$, $\mathsf{als}(e) = \{a | a \in \epsilon\}$ and $\mathsf{reg}(e) = \mathsf{rds}(e) \cup \mathsf{wrs}(e) \cup \mathsf{als}(e)$. Moreover, the set $\mathsf{mwr}(e) = \mathsf{regs}(e) \setminus \mathsf{wrs}(e)$ and the following is satisfied

\[ \mathsf{h}_{\omega_{\omega}}^{\omega} \mathsf{h'}_{\omega_{\omega}}^{\omega} \iff \forall l \in \mathsf{w}(\mathsf{rds}(e)), h \sim_{l} h'. \]

which specifies that the heaps $h$ and $h'$ are equivalent on the abstract locations $l$ in regions associated to read effects in $e$ and on the not write locations, respectively.

**Lemma 7.4.** Let $\Gamma \vdash e : \tau$ & $\epsilon$. Let $w \mathsf{w}_{\omega}^{\omega} \mathsf{w}$ be a pullback square, $\gamma \in \Gamma \mathsf{w}_{\omega}^{\omega}$ be a context, such that $\gamma : x : \chi \sim x : \chi$ and $C = (\Gamma \vdash e : \tau$ & $\epsilon \mathsf{w}(\gamma)) \in \mathcal{T}_{\omega_{\omega}}$ be computations, such that $\mu : x : \chi \sim x : \chi$. Let $h \in \mathsf{w}_{\omega}^{\omega}, h' \in \mathsf{w}$ which specifies that the heaps $h$ and $h'$ are equivalent on all the abstract locations $l$ in regions associated to read effects in $e$ and on the not write locations, respectively.

**Theorem 7.5 (Equational soundness).** Assume an ambient interpretation of types and axioms. If $\Gamma \vdash e : \tau$ & $\epsilon$ then the morphisms $\Gamma \vdash e : \tau$ & $\epsilon$ and $\Gamma \vdash e : \tau$ & $\epsilon$ are equal morphisms.

**Proof**

By induction on derivations. The core rules are direct from the category-theoretic structure identified so far. The effect-specific rules are nontrivial; however given Lemma 7.4, their proofs follow essentially the same reasoning as in existing literature [6, 41]. As for the axiom rule we argue as follows.

If $\Gamma \vdash e = v' : \tau$ using the axiom $(v, v', \tau)$ then we have $\Gamma \vdash e : \tau$ & $\epsilon$ and $\Gamma \vdash e : \tau$ & $\epsilon$ and the constant function $\lambda \gamma. p_{\tau, \tau}(\gamma)$ is continuous and thus establishes the desired equality.

---

**8. Observational equivalence**

We will now relate semantic equality to observational equivalence. So far, a trivial model that identifies everything could also have been used to "justify" our rules. Doing this, however, requires a loose relationship between the setoid interpretation and the computational meanings of raw terms which will be given by a realizability relation.

**Definition 8.1.** For each type $\tau$ & $\epsilon$, and world $w$ we define admissible relations $\Gamma \vdash e : \tau$ & $\epsilon \vdash e : \tau$ & $\epsilon$ between $\eta$ and $\epsilon$ as follows:

- $\eta \vdash e \vdash e$ for all $\eta$.
- $\eta \vdash e \vdash e$ for all $\eta$.
- $\eta \vdash e \vdash e$ for all $\eta$.
- $\eta \vdash e \vdash e$ for all $\eta$.
- $\eta \vdash e \vdash e$ for all $\eta$.

The following soundness property of $\vdash$ is proved by straightforward induction on typing derivations.

**Theorem 8.3 (Fundamental Lemma).** Assume that the interpretation of types and axioms is chosen such that for all type axioms $(\nu, \tau)$ one has $\lambda \mathsf{w}_{\omega}^{\omega} \mathsf{w}$. Then $\Gamma \vdash e : \tau$ & $\epsilon$ and $\eta \vdash e : \tau$ & $\epsilon$ if and only if $\mathsf{w}(\mathsf{w}_{\omega}^{\omega}) \mathsf{w}$. The case that tests for some observable feature of the end result signaling another in any context without compromising observable behavior.

**Definition 8.4 (Observational equivalence).** Two value expressions $v, v'$ where $v \vdash v' : \tau$ & $\epsilon$ and $\eta \vdash e \vdash e$ are observationally equivalent at type $\tau$ if for all $f$ such that $f v = f v'$ it holds that $\mathsf{w}(\mathsf{w}_{\omega}^{\omega}) \mathsf{w}$.

It is easy to extend this definition to open terms and computation expressions by closing up with lambda abstractions. Intuitively, observationally equivalent expressions, can be replaced by one another in any context without compromising observable behavior. To see this, one can build observations $f$ from such a context by first $\lambda$-abstracting its hole and then adding at the end a wrapper function that tests for some observable feature of the end result signaling failure of the test by nontermination.

**Theorem 8.5 (main result).** If $v \vdash v' : \tau$ & $\epsilon$ then $v$ and $v'$ are observationally equivalent.

**Proof**

If $f$ is an observation then by the congruence rules we get $f v = f v'$ & $\tau$ so, by Theorem 7.5 we find $\mathsf{w}(\mathsf{w}_{\omega}^{\omega}) \mathsf{w}$. Thus in particular $\mathsf{w}(\mathsf{w}_{\omega}^{\omega}) \mathsf{w}$ is empty world, empty heap.
defined iff \( \forall f : f \not\in \emptyset \) is defined. Now, if \( \forall f : f \not\in \emptyset \) is defined then by Theorem 8.3 and the definition of \( \triangleright \) we can conclude that \( \forall f : f \not\in \emptyset \) is defined so, using Theorem 8.3 again we get that \( \forall f : f \not\in \emptyset \) is defined. \( \square \)

9. Applications

In this section we give some concrete instantiations of our framework in the form of axioms and their semantic justifications. We do so in more detail for the Overlapping References example shown in Section 2, and in less detail for the other examples.

Overlapping References

Recall value \( v_{\omega} \) and type \( \tau_{\omega} \) from Section 2 (two overlapping references). In order to justify the axioms \( (v_{\omega}, \tau_{\omega}) \) we have to construct a semantic object \( x_{\omega} \in \| \tau_{\omega} \| \) such that \( v_{\omega} \triangleright x_{\omega} \) which we will now do.

Given world \( w \) we put \( x_{\omega}(w)(\bullet) = (c_0, c_1) \), where \( c_0 \) and \( c_1 \) are defined as follows \((\bullet \in \text{unit}(w))\): 

\[
c_0(h) = (\lambda w : w \rightarrow h_1, (g_1, g_2, s_1, s_2))
\]

where \( \text{new}(h, 0) = (X, h_1) \), that is, \( h_1 \) is obtained from \( h \) by allocating a concrete location \( X \); \( w \) is the world obtained by extending \( w \) with the abstract locations \( s_1, g_1 \) and \( s_2, g_2 \) defined in Section 4.1, where the former is marked with region \( r_l \) and the latter with region \( r_r \); finally, \( (g_1, g_2, s_1, s_2) \) are the corresponding semantic objects that get and set values to the projections of pairs stored in \( X \), e.g., \( g_1 \in \| \text{unit} \| \| \text{int} \| w \) is the semantic counterpart of the function getting the first projection of the pair stored in \( X \), defined in a similar way as we do for \( x_{\omega} \).

Note that \( v_{\omega} \triangleright x_{\omega} \) in particular requires that the new heap returned by \( c_0 \) agrees with the one returned by \( v_{\omega} \), and, more generally, we are forced to follow the computational behavior of \( v_{\omega} \) in \( x_{\omega} \). We are free, though to choose the semantic components such as world extensions and, later, pullbacks.

The object \( c_1 \), now, is defined as follows: Assume that \( R \in \mathcal{R}(\mathcal{A}(\mathcal{L}(\mathcal{A}(), \mathcal{A}()))) \) is a heap relation and two heaps \( h, h' \in \mathcal{H} \) that are related, i.e., \( (h, h') \in R \). Moreover, assume that \( c_0(h) = (\lambda w : w \rightarrow w \& s_1 \& h_1, (g_1, g_2, s_1, s_2)) \) and \( c_0(h') = (\lambda w : w \rightarrow w \& s_2 \& h_1', (g_1', g_2', s_1', s_2')) \) where the former allocates \( X \), while the latter allocates \( X' \). Note that everything terminates in this example. Now \( c_1(R, h, h') \) returns the pullback \( w_1 \triangleright x_{\omega} = \mathcal{X} \mathcal{X}_{s_1, s_2} \mathcal{X}_{g_1, g_2} w_1 \), where \( X \) is mapped to \( X' \). It is easy to check that \( h_1 \) and \( h_1' \) are related in \( R \) when taken to the pullback's low point and that \( (g_1, g_2, s_1, s_2) \) and \( (g_1', g_2', s_1', s_2') \) are equal in its apex.

Note that once this typing has been justified all equations derivable from it are automatically sound for observational equivalence. Thus, in particular, writes to the two "overlapping reference" composite.

Set factory

Let \( w \) be a world and \( h \in \mathcal{H} \). Suppose that \( h_1 \) arises from \( h \) by allocating a fresh set data structure, \( e \), a linked list, with entry point \( X \). Let \( set_e \) be the abstract location describing this fresh data structure.

Now for any chosen region \( t \) we can add \( set_t \) to \( t \) to yield a new world \( w_t \). The function \( \text{setfactory}_R(w)(h) \) then returns \( w_t \) and a tuple of semantic functions for reading, membership, removal, of which we only sketch reading here: If \( t : w_1 \rightarrow w_2 \) and \( h_1 \in \mathcal{H}(w_2) \) and \( i \in \mathcal{H}(w_1) \) then the reading function looks up \( i \) in the data structure starting at the entry points \( X \) in \( h_1 \) (note that \( h_1 \in \mathcal{H}(w) \) asserts that this data structure exists and is well-formed. The returned (abstract) store \( h_2 \) might not be the same as \( h \) because internal reorganizations, \( e \), removal of duplicates, might have occurred, but no world extension is needed and \( h_1 \sim h_2 \). This holds together with the fact that the outcome only depends on the set equivalence class (in the proof-relevant sense) justifies a read-only typing for reading.

Memoization

For the simple memo functional from Section 2 we produce just as in the previous example a fresh abstract location \( 1 \) whose specification looks only at the two newly allocated concrete locations, say \( l_1, l_2 \), where the first stores the argument value \( i \) and the second the integer value \( f(i) \) where \( f \) is the pure function to be memoised. The location’s \( l \) rely specifies that two heaps are equivalent when the values stored in \( l_1 \) and \( l_2 \) are the same: \( (h, l_1)' \equiv (h, l_2)' \Leftrightarrow h(l_1) = h(l_2) \). We see in Lemma 7.4 that if a function is semantically pure (empty effect) then there is a world- and heap-independent function describing its action. Thus the memo functional may be considered pure.

9.1 State Dependent Abstract Data Types

Although our focus is on effect-dependent equivalences that can be derived from refined typings using our equational rules, we are able to justify some of the well-known tricky examples from the literature on proving equivalences in ML-like languages without effect annotations. Our guarantee conditions here act like the transition system components of the worlds used by Ahmed et al. [1, 19].

Awkward Example

The first example is the classic awkward example [34]. Consider the following two programs:

\[
e_1 = \text{let } x := \text{ref}(0) \text{ in } \text{if } x = 1 \text{ then } f(); !x \text{ and } e_2 = \text{let } f() = 1.1
\]

Intuitively, \( e_1 \) and \( e_2 \) are equivalent as they both return the value \( 1 \), although \( e_1 \) uses a fresh location to do so. We can formally prove the equivalence as follows: Assign the region where \( x \) is allocated as \( r \). If \( f \) has the type unit \( \rightarrow \) unit with effects \( e \), then \( e_1 \) has type (unit \( \rightarrow \) unit) \( \rightarrow \) int & \( a_l \), while \( e_2 \) has type (unit \( \rightarrow \) unit) \( \rightarrow \) int & \( e \). Notice that \( e \) may contain \( a_r \) or \( w_r \). Moreover, assume that an abstract location in region \( r \) looks at a single concrete location \( l \):

\[
l' = (h, l_1) | l | l' \neq 1 \text{ and } l' = h(l_1) \cup \{(h, h) | h : l'\}
\]

Notice that it is correct to assign \( e_1 \) a write effect in region \( r \), as it writes the value \( 1 \) to the location assigned to \( x \).

For proving the equivalence of \( e_1 \) and \( e_2 \), assume a world \( w \) and a heap \( h \). Let \( \epsilon \| \text{wh} = (w \cup \{h \}) \) and \( \epsilon \| \text{wh} = (w \cup \{h \}) \). We construct a pullback square \( w \cup \{h \} \cup \{a_l \} \cup \{a_r \} \cup \{w_r \} \cup \{w_r \} \cup \{w_r \} \cup \{w_r \} \). Similarly, \( h \) when taken to the low point of the square, that is, where the locations in \( w_r \) are forgotten, the resulting heap is equivalent to \( h_2 \).

Modified Awkward Example

Consider now the following variant [1, 19] of the awkward example:

\[
e_1 = \text{let } x := \text{ref}(0) \text{ in } \text{if } f(x) = 0 \text{ then } f(); x \text{ and } e_2 = \text{let } f() = 1.1
\]

The difference is that in the first program \( x \) is first assigned \( 0 \) and the call-back function is used twice. Interestingly, however, our solution for the Awkward example still works just fine. We can prove semantically that the type of the program \( e_1 \) has the same type as before in the Awkward example, where the only writes allowed on the abstract location assigned for \( x \) is to write the concrete location to one. Then the reasoning comes in a similar way.

Callback with Lock Example

We prove the equivalence of the following programs [1, 19]:

```r
f(x)
```
\[ e_1 = \text{let } b \equiv \text{ref}(\text{true}) \text{ in let } x \equiv \text{ref}(0) \text{ in } \langle \lambda f. \text{if } b \text{ then } (b \equiv \text{false}; f(0); x \equiv 1 + 1; b \equiv \text{true} \rangle \rangle \]

\[ e_2 = \text{let } b \equiv \text{ref}(\text{true}) \text{ in let } x \equiv \text{ref}(0) \text{ in } \langle \lambda f. \text{if } b \text{ then } (b \equiv \text{false}; \text{let } n \equiv 1 \text{ in } f(0); x \equiv n + 1; b \equiv \text{true} \rangle \rangle \]

Both programs produce a pair of functions, one that increments the value stored in \( x \) and the second that returns the value stored in \( x \). They use the boolean reference \( b \) as lock in the incrementing function. In particular, once this function is called the value in \( b \) is set to \text{false} and only after the call-back is called and the value in \( x \) is incremented is \( b \) set again to \text{true}. However, the implementation of the increment function is different. While \( e_1 \) invokes the call-back \( f(0) \) and then increments the value in \( x, e_2 \) remembers (in \( n \)) the value of \( x \) before the call-back and then uses \( n \) to update the value stored in \( x \).

Assume that \( x \) and \( b \) are allocated in the same location (\( l \)) in the region \( r \). We show that these programs are equivalent in the type \((\text{unit} \rightarrow \text{unit}) \rightarrow \text{unit} \times \text{unit} \) & \( r \), where \( e \) may contain the effects \( w_r, rd \). In particular, the location \( l \) looks at two concrete locations \( l_b \) and \( l \); (storing \( x \) and \( b \)):

\[
\begin{align*}
\ell^b & = \langle (h, h') \mid b(h) = h' \rangle \\
\ell^f & = \langle (h, h') \mid b(h) = \text{false} \Rightarrow h_1 = h \text{ and } (h_0, h) = \text{true} \rangle
\end{align*}
\]

First, notice that indeed the two programs above have the effect \( w_r \). The increment of \( x \) is allowed by \( \ell^f \), as \( b \) is initially \text{true}. To show that the two programs above are equivalent, we show that the value stored in \( x \) before and after the call back is called remains the same. This is the case, as even if the callback function has a write \( e \), it cannot change the value of \( x \). This is because when the function is called, the value stored in \( b \) is \text{false}, meaning that the heap remains the same.

Despite the above, the model as presented here does not directly justify equivalences involving representation independence, \( e \), that our overlapping references \( v_w \) are observationally equivalent to an implementation that really allocates two integer references. Generalizing world morphisms as discussed after Def. 6.2 does allow many such equivalences to be proved, though a parametric variant, along the lines of that described by Stark [39], would be more powerful still.

10. Related and Future Work

We have shown Proof-Relevant Logical Relations, introduced in our previous work [4], can be used to justify nontrivial effect-dependent program equivalences. For the first time it was possible to combine effect-dependent program equivalences with hidden invariants allowing “silent modifications” that do not count towards the ascription of an effect. Earlier accounts of effect-dependent program equivalences [3, 6, 7, 26, 41] do not provide such possibilities.

Bisimulation

An alternative approach to proving ‘difficult’ contextual equivalences is to use techniques based on bisimulation [27]. While bisimulation has been applied to typed calculi [24, 40] the strength of the method lies in being able to deal with fancy computation rules and fairly simple types and contracts rather than well-understood computation rules (call-by-value lambda calculus in our case) and fancy types and contracts. Indeed, we believe that something like our abstract locations and the proof-relevant world extensions would also suggest itself in a bisimulation-based approach to the equivalences studied here.

Model Variables

Proof-relevant logical relations or rather the sets [\( AW \)] where \( A \) is a semantic type bear a vague relationship with the model variables [15] from “design by contract” [32] and more generally data refinement [16]. The commonality is that we track the semantic behavior of a program part with abstract functions on some abstracted set of data that may contain additional information (the “model”). The difference is that we do not focus on particular proof methods or specification formalisms but that we provide a general, sound semantic model for observational equivalence and program transformation and not merely for functional correctness. This is possible by the additional, also proof-relevant part of the semantic equality proofs between the elements of the models. We also note that our account rigorously supports higher-order functions, recursion, and dynamic allocation.

Separation Logic

Our abstract locations draw upon several ideas from separation logic [37], in particular the conditions on rely/guarantee assumptions from [42]. Intriguingly, we did not need something resembling the ‘frame rule’ although perhaps the II-quantification over larger worlds in function spaces plays its role.

Our use of pullback-preserving functors is motivated by FM-sets [21] or rather the Schanuel topos to which they are equivalent. Pitts’s [35] gives a comprehensive account. The way in which we work with both permutation actions and functors has some precedent [9], but does feel slightly awkward: it really ought to be possible to combine the two structures into one.

We would like to define a stylized format that allows one to discharge semantic proof obligations in some cases without having to go down to the low-level semantic definitions. We noted that private transitions [1] serve a similar purpose and might perhaps suggest a possible approach in our case, too.

We would also like to be able to store values with proof-relevant equality, which would allow a stratified form of higher-order store [12]. Unrestricted higher-order store requires circular definitions of worlds, which has previously been addressed using, for example, metric spaces [11].

Acknowledgments

We thank Lennart Beringer and Andrew Kennedy for fruitful discussions. Nigam was supported by CAPES / CNPq and DAAD.

References


manitics for effect-based program transformations: higher-order store. In PPDP, 2009.

tions for name generation. In TLCA, 2013.


manitics for effect-based program transformations with dynamic al-


semantics for storage. In TLCA, volume 3461 of LNCS, 2005.


solution of recursive metric-space equations. Theor. Comp. Sci.,

[12] G. Boudol. Typing termination in a higher-order concurrent imperative


