Abstract Effects and Proof-Relevant Logical Relations

Nick Benton
Microsoft Research, Cambridge, UK
nick@microsoft.com

Martin Hofmann
LMU, Munich, Germany
hofmann@ifi.lmu.de

Vivek Nigam
UFPB, João Pessoa, Brazil
vivek.nigam@gmail.com

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 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 unrefined 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-

POPL '14, January 22–24, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2535838.2535869

lence relations. This allows, for example, a lookup function for a set ADT to be assigned a read-but-not-write effect, even though the concrete implementation may involve non-observable writes to rebalance an internal datastructure. Moving to PERs in this way requires us to revisit the notion of separation, allowing types to involve distinct regions whose concrete implementations overlap, albeit non-observably, in memory.

Earlier models of dynamic allocation [6] have used Kripke logical relations in which worlds are finite partial bijections between locations, with region-colored links. Two computations $c, c' : \mathbb{H} \rightharpoonup \mathbb{H} \times \mathbb{V}$, where $\mathbb{H}, \mathbb{V}$ are sets of (partial) heaps and values, respectively, are in the computation relation $T_\varepsilon[\![A]\!]\mathsf{w}$, for world w, effect ε and result type A when $\forall \mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1.\ \mathsf{h}, \mathsf{h}' \models \mathsf{w} \Rightarrow \exists \mathsf{w}_1 \geq \mathsf{w}.\ \mathsf{h}_1, \mathsf{h}'_1 \models \mathsf{w}_1 \wedge (\mathsf{h}, \mathsf{h}_1, \mathsf{h}', \mathsf{h}'_1, \mathsf{w}, \mathsf{w}_1) \in [\![\varepsilon]\!] \wedge (v, v') \in [\![A]\!]\mathsf{w}_1$ where $c\,\mathsf{h} = (\mathsf{h}_1, v)$ and $c'\,\mathsf{h}' = (\mathsf{h}'_1, v')$. Here, $[\![A]\!]\mathsf{w}_1$ is the (world-dependent) logical relation at the result type and $[\![\varepsilon]\!]$ the interpretation of the effect, which can be given in various ways, *e.g.*, in terms of binary relations on stores to be preserved. The notation $\mathsf{h}, \mathsf{h}' \models \mathsf{w}$ means that for each link $(\mathsf{l}, \mathsf{l}') \in \mathsf{w}$, $\mathsf{l} \in \mathrm{dom}(\mathsf{h}), \mathsf{l}' \in \mathrm{dom}(\mathsf{h}')$.

However, generalizing such a logical relation from bijections to PERs is surprisingly difficult. The problematic part is the existential quantification over world extensions: $\exists \mathsf{w}_1 \geq \mathsf{w}$. This says that c and c' are related at world w if there is *some* future world w_1, allowing for the computations to perform allocations, at which the respective final states and values are related. This pattern of quantification occurs in many accounts of generativity, but the dependence of w_1 on both h and h' creates a difficulty when one generalizes from bijections to PERs and tries to prove equivalences. Roughly, one has to consider varying the initial heap in which one computation, say c', is started; the existential then supplies one with a *different* extension that is not at all related, even on the side of c where the heap stays the same, to the one with which one started. In the case of bijections, the fact that h_1 depends only on h (not on h') allows one to deduce sufficient information about the domain of w_1 from the clause $\mathsf{h}_1, \mathsf{h}'_1 \models \mathsf{w}_1$, but this breaks down in the more abstract setting.

To fix this problem, we will essentially replace the existential quantifier in the logical relation by appropriate Skolem functions, explicitly enforcing the correct dependencies. In the language of type theory, this amounts to replacing the existential with a Σ-type. A statement like $(c, c') \in T_\varepsilon[\![A]\!]$ is no longer just a proposition, but we rather have a "set of proofs" $(p : c \sim c') \in T_\varepsilon[\![A]\!]$, and our constructions will explicitly maps proofs to proofs.

We have previously shown [4] how the formalism of *setoids* can be used to make such intuitions both rigorous and more general. That work, dealing with a version of Pitts and Stark's ν-calculus, used proof-relevant setoids to solve another problem associated with the use of existential quantification: that it fails to preserve admissibility of predicates and relations and so interacts badly with general recursion. Here we show how that technology scales to a much richer language and type system.

2. Motivating Examples

We will assign effect-refined types to terms in a fairly conventional monadically-structured untyped metalanguage with higher-order functions and dynamically allocated references to flat data. The language is defined formally later on, but the following examples of the kind of equivalences and typings we will be able to justify should be comprehensible.

Memoisation Let *memo* be the function

$$\lambda f.\mathtt{let}\ x \Leftarrow \mathtt{ref}(\underline{0})\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow \mathtt{ref}(f\ \underline{0})\ \mathtt{in}\\ \lambda a.\mathtt{if}\ a = !x\ \mathtt{then}\ !y\ \mathtt{else}\ \mathtt{let}\ r \Leftarrow f\ a\ \mathtt{in}\ x := a; y := r; r$$

where $t_1; t_2 = \mathtt{let}\ \_ \Leftarrow t_1\ \mathtt{in}\ t_2$ is sequential composition and $=$ is equality on storable values. So, *memo* returns a memoized version of its argument. Unlike previous models of effects, our model justifies the typing $memo : (int \xrightarrow{\emptyset} int) \xrightarrow{\emptyset} (int \xrightarrow{\emptyset} int)$, saying that if f is observationally pure, *memo* f, is too, 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:

$$\lambda y.\mathtt{let}\, g \Leftarrow memo\,(\lambda x.x+1)\,\mathtt{in}\, e \equiv \mathtt{let}\, g \Leftarrow memo\,(\lambda x.x+1)\,\mathtt{in}\,\lambda y.e$$

Overlapping references Let p, p^{-1} implement a bijection $\mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$, then

$$\begin{aligned} v_{or} := \lambda\_.\mathtt{let}\ r \Leftarrow \mathtt{ref}(\underline{0})\ \mathtt{in}\ (\lambda\_.(p(!r)).1,\ \lambda\_.(p(!r)).2,\\ \lambda n.\mathtt{let}\ (x,y) \Leftarrow p(!r)\ \mathtt{in}\ r := p^{-1}(n,y),\\ \lambda n.\mathtt{let}\ (x,y) \Leftarrow p(!r)\ \mathtt{in}\ r := p^{-1}(x,n)) \end{aligned}$$

which multiplexes two abstract integer references onto a single concrete one, can be given the type

$$\begin{aligned} \tau_{or} := \quad & unit \xrightarrow{al_{\mathsf{r}1}, al_{\mathsf{r}2}} (unit \xrightarrow{rd_{\mathsf{r}1}} int) \times (unit \xrightarrow{rd_{\mathsf{r}2}} int) \times \\ & (int \xrightarrow{wr_{\mathsf{r}1}} unit) \times (int \xrightarrow{wr_{\mathsf{r}2}} unit) \end{aligned}$$

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 *setfactory* that generates mutable integer sets. Calling *setfactory* allocates a new reference cell pointing to a linked list of integers (initially empty) and returns a triple of functions (mem, add, 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 *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 *setfactory*:

$$setfactory : \forall \mathsf{r}.unit \xrightarrow{al_{\mathsf{r}}} (int \xrightarrow{rd_{\mathsf{r}}} bool) \times (int \xrightarrow{rd_{\mathsf{r}}, wr_{\mathsf{r}}} unit) \times (int \xrightarrow{rd_{\mathsf{r}}, wr_{\mathsf{r}}} unit)$$

which expresses that *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, *mem* operations may be swapped and duplicated.

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

$$\begin{aligned} make &= \lambda\_.\mathtt{ref}(\underline{0})\\ getx &= \lambda p.\mathtt{if}\ !p = \underline{0}\ \mathtt{then}\ \underline{0}\ \mathtt{else}\ !(!p.1)\\ gety &= \lambda p.\mathtt{if}\ !p = \underline{0}\ \mathtt{then}\ \underline{0}\ \mathtt{else}\ !(!p.2)\\ setx &= \lambda p.\lambda n.\mathtt{if}\ !p = \underline{0}\ \mathtt{then}\ p := (\mathtt{ref}(n), \mathtt{ref}(\underline{0}))\ \mathtt{else}\ !p.1 := n\\ sety &= \lambda p.\lambda n.\mathtt{if}\ !p = \underline{0}\ \mathtt{then}\ p := (\mathtt{ref}(\underline{0}), \mathtt{ref}(n))\ \mathtt{else}\ !p.2 := n \end{aligned}$$

We can introduce a new abstract type $point_{\mathsf{r}1,\mathsf{r}2}$ for which the following typings can be semantically justified:

$$\begin{array}{ll} make : unit \xrightarrow{al_{\mathsf{r}2}} point_{\mathsf{r}1,\mathsf{r}2} & \\ getx : point_{\mathsf{r}1,\mathsf{r}2} \xrightarrow{rd_{\mathsf{r}1}} int & gety : point_{\mathsf{r}1,\mathsf{r}2} \xrightarrow{rd_{\mathsf{r}2}} int \\ setx : point_{\mathsf{r}1,\mathsf{r}2} \xrightarrow{\emptyset} int \xrightarrow{wr_{\mathsf{r}1}} unit & setx : point_{\mathsf{r}1,\mathsf{r}2} \xrightarrow{\emptyset} int \xrightarrow{wr_{\mathsf{r}2}} unit \end{array}$$

Observe that the allocation effects are flagged in the *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 ω-cpo, *i.e.*, a partial order with suprema of ascending chains. A *domain* is a predomain with a least element, $\bot$. 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.*, $f(\sup_i x_i) = \sup_i f(x_i)$. 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 $(a_i)_i$ is an ascending chain in A such that $a_i \in U$ for all i, then $\sup_i a_i \in U$, too. If $f : X \times A \to A$ is continuous and A is a domain then one defines $f^\dagger(x) = \sup_i f_x^i(\bot)$ with $f_x(a) = f(x, a)$. One has, $f(x, f^\dagger(x)) = f^\dagger(x)$ and if $U \subseteq A$ is admissible and $f : X \times U \to U$ then $f^\dagger : X \to U$, too. We denote a partial (continuous) function from set (predomain) A to set (predomain) B by $f : A \rightharpoonup B$.

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

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

Note that heaps and R-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 $\mathbb{V}$ and the domain of computations $\mathbb{C}$ simultaneously as follows: $\mathbb{C} = \mathbb{H} \rightharpoonup \mathbb{H} \times \mathbb{V}$ are partial continuous functions from $\mathbb{H}$ to $\mathbb{H} \times \mathbb{V}$, 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: $\mathbb{V} \simeq \mathbb{V}_b + \mathit{fun}(\mathbb{V} \to \mathbb{C}) + \mathbb{V}^*$. Such domain equations can be solved by standard methods.

Syntax The syntax of untyped values and computations is:

$$\begin{array}{lcl} v & ::= & x \mid () \mid c \mid (v_1, v_2) \mid v.1 \mid v.2 \mid \mathtt{rec}\ f\ x = t \\ t & ::= & v \mid \mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2 \mid v_1\ v_2 \mid \mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \\ & & !v \mid v_1 := v_2 \mid \mathtt{ref}(v) \end{array}$$

Here, x ranges over variables and constants c over constant symbols, each of which has an associated interpretation $[\![c]\!] \in \mathbb{V}$; these include numerals $\underline{n}$ with $[\![\underline{n}]\!] = int(n)$, 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. $\mathtt{rec}\ f\ x = t$ 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 $f \notin fv(t)$. Finally, $!v$ (reading) returns the contents of location v, $v_1 := v_2$ (writing) updates location v_1 with value v_2, and $\mathtt{ref}(v)$ (allocating) returns a fresh location initialized with v. The metatheory 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 $[\![v]\!] \in \mathbb{V} \to \mathbb{V}$ and terms $[\![t]\!] \in \mathbb{V} \to \mathbb{C}$ is defined in usual way by inductive clauses, which we omit here.

Types are given by the grammar

$$\tau ::= \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid A \mid \tau_1 \times \tau_2 \mid \tau_1 \xrightarrow{\varepsilon} \tau_2$$

where A ranges over user-specified abstract types. They will typically include region-indexed reference types such as $\mathtt{intref}_\mathsf{r}$ and also types like lists, sets, and even objects, again possibly refined by regions. The metavariable ε 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 Dead requires us to ascertain that t_1 terminates for all inputs and heaps satisfying the contracts specified in Γ, ε.

We assume an ambient set of *typing axioms* each having the form (v, τ) where v is a value in the metalanguage and τ is a type meaning that v is claimed to be of type τ and that this will be proved "manually" using the semantics rather than using the typing rules.

We also assume an ambient set of *equality axioms* each having the form (v, v', τ) and asserting semantic equality of v and v' at type τ, again to be justified "manually". We assume that whenever (v, v', τ) is an axiom so are (v, τ) and (v', τ).

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 `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

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \underline{n} : \texttt{int}} \qquad \frac{(v,\tau) \text{ a type axiom}}{\Gamma \vdash v : \tau} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \tau \,\&\, \emptyset}$$

$$\frac{\Gamma \vdash v : \tau_1 \times \tau_2}{\Gamma \vdash v.i : \tau_i} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \xrightarrow{\varepsilon} \tau_2 \quad \Gamma \vdash v_2 : \tau_1}{\Gamma \vdash v_1\ v_2 : \tau_2 \,\&\, \varepsilon} \qquad \frac{\Gamma, f{:}\tau_1 \xrightarrow{\varepsilon} \tau_2, x{:}\tau_1 \vdash e : \tau_2 \,\&\, \varepsilon}{\Gamma \vdash \texttt{rec}\ f\ x = e : \tau_1 \xrightarrow{\varepsilon} \tau_2} \qquad \frac{\Gamma \vdash e : \tau \,\&\, \varepsilon_1 \quad \varepsilon_1 \subseteq \varepsilon_2}{\Gamma \vdash e : \tau \,\&\, \varepsilon_2}$$

$$\frac{\Gamma \vdash v : \texttt{int} \quad \Gamma \vdash e_1 : \tau \,\&\, \varepsilon \quad \Gamma \vdash e_2 : \tau \,\&\, \varepsilon}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \,\&\, \varepsilon} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \,\&\, \varepsilon \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2 \,\&\, \varepsilon}{\Gamma \vdash \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2 : \tau_2 \,\&\, \varepsilon} \qquad \frac{\Gamma \vdash t : \tau \,\&\, \varepsilon \quad \mathsf{r} \notin \text{regs}(\Gamma) \cup \text{regs}(\tau)}{\Gamma \vdash t : \tau \,\&\, \varepsilon \setminus \{rd_{\mathsf{r}}, wr_{\mathsf{r}}, al_{\mathsf{r}}\}}\ \text{Masking}$$

Figure 1. Core rules for effect typing

Basic Equations (Extract)

$$\frac{\Gamma \vdash t : \tau \,\&\, \varepsilon}{\Gamma \vdash t = t : \tau \,\&\, \varepsilon} \qquad \frac{\Gamma \vdash t = t' : \tau \,\&\, \varepsilon}{\Gamma \vdash t' = t : \tau \,\&\, \varepsilon} \qquad \frac{\Gamma \vdash t = t' : \tau \,\&\, \varepsilon \quad \Gamma \vdash t' = t'' : \tau \,\&\, \varepsilon}{\Gamma \vdash t = t'' : \tau \,\&\, \varepsilon} \qquad \frac{\Gamma \vdash v = v' : \tau}{\Gamma \vdash v = v' : \tau \,\&\, \emptyset} \qquad \frac{\Gamma, \vdash v : \tau_1 \times \tau_2}{\Gamma \vdash v = (v.1, v.2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2).i = v_i : \tau_i} \qquad \frac{\Gamma, f : \tau_1 \xrightarrow{\varepsilon} \tau_2, x{:}\tau_1 \vdash t = t' : \tau_2 \,\&\, \varepsilon}{\Gamma \vdash (\texttt{rec}\ f\ x = t) = (\texttt{rec}\ f\ x = t') : \tau_1 \xrightarrow{\varepsilon} \tau_2} \qquad \frac{\Gamma, f : \tau_1 \xrightarrow{\varepsilon} \tau_2, x{:}\tau_1 \vdash t : \tau_2 \,\&\, \varepsilon \quad \Gamma \vdash v : \tau_1}{\Gamma \vdash (\texttt{rec}\ f\ x = t)\, v = t[v/x, (\texttt{rec}\ f\ x = t)/f] : \tau_2 \,\&\, \varepsilon}$$

$$\frac{\Gamma \vdash v : \tau_1 \,\&\, \varepsilon \quad \Gamma, x : \tau_1 \vdash t : \tau_2 \,\&\, \varepsilon}{\Gamma \vdash \texttt{let}\ x \Leftarrow v\ \texttt{in}\ t = t[v/x] : \tau_2 \,\&\, \varepsilon} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \,\&\, \varepsilon \quad \Gamma \vdash t_2 : \tau_2 \,\&\, \varepsilon \quad \Gamma, x : \tau_2, y : \tau_1 \vdash t_3 : \tau_3 \,\&\, \varepsilon}{\Gamma \vdash \texttt{let}\ x \Leftarrow (\texttt{let}\ y \Leftarrow t_1\ \texttt{in}\ t_2)\ \texttt{in}\ t_3 = \texttt{let}\ y \Leftarrow t_1\ \texttt{in let}\ x \Leftarrow t_2\ \texttt{in}\ t_3 : \tau_3 \,\&\, \varepsilon} \qquad \frac{(v, v', \tau) \text{ an equality axiom}}{\Gamma \vdash v = v' : \tau}$$

Congruence Equations (Extract)

$$\frac{\Gamma \vdash v = v' : \tau_1 \times \tau_2}{\Gamma \vdash v.i = v'.i : \tau_i} \qquad \frac{\Gamma \vdash v = v' : \texttt{bool} \,\&\, \emptyset \quad \Gamma \vdash t_1 = t_1' : \tau \,\&\, \varepsilon \quad \Gamma \vdash t_1 = t_1' : \tau \,\&\, \varepsilon}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2 = \texttt{if}\ v'\ \texttt{then}\ t_1'\ \texttt{else}\ t_2' : \tau \,\&\, \varepsilon}$$

Effect Based Equations

$$\frac{\Gamma \vdash t_1 : \tau_1 \,\&\, \varepsilon_1 \quad \Gamma \vdash t : \tau \,\&\, \varepsilon \quad \text{wrs}(\varepsilon_1) = \emptyset}{\Gamma \vdash (\texttt{let}\ x \Leftarrow t_1\ \texttt{in}\ t) = t : \tau \,\&\, \varepsilon}\ \text{Dead, provided } t_1 \text{ terminates}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \,\&\, \emptyset \quad \Gamma, x : X, y : \tau_1 \vdash t : \tau \,\&\, \varepsilon}{\Gamma \vdash (\lambda x.\texttt{let}\ y \Leftarrow t_1\ \texttt{in}\ t) = (\texttt{let}\ y \Leftarrow t_1\ \texttt{in}\ \lambda x.t) : (X \xrightarrow{\varepsilon} \tau) \,\&\, \emptyset}\ \text{Pure Lambda Hoist}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \,\&\, \varepsilon_1 \quad \Gamma, x : \tau_1, y : \tau_1 \vdash t : \tau \,\&\, \varepsilon \quad \text{wrs}(\varepsilon_1) \cap \text{rds}(\varepsilon_1) = \text{als}(\varepsilon_1) = \emptyset}{\Gamma \vdash (\texttt{let}\ x \Leftarrow t_1\ \texttt{in let}\ y \Leftarrow t_1\ \texttt{in}\ t) = (\texttt{let}\ x \Leftarrow t_1\ \texttt{in}\ t) : \tau \,\&\, \varepsilon \cup \varepsilon_1}\ \text{Duplicated}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \tau_1 \,\&\, \varepsilon_1 \quad \Gamma \vdash t_2 : \tau_2 \,\&\, \varepsilon_2 \quad \Gamma, x : \tau_1, y : \tau_2 \vdash t : \tau \,\&\, \varepsilon \\ \text{rds}(\varepsilon_1) \cap \text{wrs}(\varepsilon_2) = \text{rds}(\varepsilon_2) \cap \text{wrs}(\varepsilon_1) = \text{wrs}(\varepsilon_1) \cap \text{wrs}(\varepsilon_2) = \emptyset\end{array}}{\Gamma \vdash (\texttt{let}\ x \Leftarrow t_1\ \texttt{in let}\ y \Leftarrow t_2\ \texttt{in}\ t) = (\texttt{let}\ y \Leftarrow t_2\ \texttt{in let}\ x \Leftarrow t_1\ \texttt{in}\ t) : \tau \,\&\, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon}\ \text{Commuting}$$

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

$$\pi \cdot \mathsf{l} = \pi(\mathsf{l}) \qquad \pi \cdot (a_1, a_2) = (\pi \cdot a_1, \pi \cdot a_2)$$
$$\pi \cdot (L \subseteq \mathbb{L}) = \{\pi(\mathsf{l}) \mid \mathsf{l} \in L\} \qquad \text{dom}(\pi \cdot \mathsf{h}) = \pi \cdot (\text{dom}(\mathsf{h}))$$
$$(\pi \cdot \mathsf{h})(\mathsf{l}) = \pi \cdot (\mathsf{h}(\pi^{-1}(\mathsf{l}))) \qquad \pi \cdot R = \{\vec{\mathsf{h}} \mid \pi^{-1} \cdot \vec{\mathsf{h}} \in R\}$$
$$\pi \cdot (\mathsf{h}_1, \ldots, \mathsf{h}_n) = (\pi \cdot \mathsf{h}_1, \ldots, \pi \cdot \mathsf{h}_n)$$

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 $perm(\mathbb{L})$ for the group of permutations of $\mathbb{L}$ and define an action $\pi \cdot -$ of permutations $\pi \in perm(\mathbb{L})$ for values, stores, sets of locations, and relation on stores. Table 3 contains the most important cases: Note that permutations act differently on sets of locations and on sets / 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 $\vec{\mathsf{h}}$ is in the relation $\pi \cdot R$ iff the list of heaps obtained by undoing the permutation $(\pi^{-1} \cdot \vec{\mathsf{h}})$ is in the relation R. We define the set of fixpoints of a permutation π as $\text{fix}(\pi) = \{\mathsf{l} \mid \pi(\mathsf{l}) = \mathsf{l}\}$. 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) *least* support, which we write $supp(x)$.

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 $\mathsf{h} \in P$ 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 $\mathfrak{l}$ *(on the chosen set $\mathbb{H}$) consists of the following data:*

- *a non-empty, finitely supported, partial equivalence relation (PER) $\mathfrak{l}^R$ on $\mathbb{H}$ modelling the "semantic equality" on the bits of the store that $\mathfrak{l}$ uses (a "rely-condition");*
- *a finitely supported transitive relation $\mathfrak{l}^G$ modelling what bits of the store "writes to $\mathfrak{l}$" leave intact (a "guarantee condition").*

That is, if $(\mathsf{h}, \mathsf{h}_1) \in \mathfrak{l}^G$ *then* h_1 *might arise by writing to* $\mathfrak{l}$ *in* h *and all possible writes are specified by* $\mathfrak{l}^G$*;*

subject to the conditions, where π *is a permutation and* $\mathsf{h} : \mathfrak{l}^R$ *is shorthand for* $(\mathsf{h}, \mathsf{h}) \in \mathfrak{l}^R$*:*

1. *if* $\mathrm{fix}(\pi) \supseteq supp(\mathfrak{l}^R)$ *then* $\mathsf{h} : \mathfrak{l}^R \Rightarrow (\mathsf{h}, \pi \cdot \mathsf{h}) \in \mathfrak{l}^R$.
2. *if* $\mathsf{h} : \mathfrak{l}^R$ *then* $(\mathsf{h}, \mathsf{h}) \in \mathfrak{l}^G$*;*
3. *if* $(\mathsf{h}, \mathsf{h}_1) \in \mathfrak{l}^G$ *then* $\mathsf{h} : \mathfrak{l}^R$ *and* $\mathsf{h}_1 : \mathfrak{l}^R$*;*
4. $\mathfrak{l}^R; \mathfrak{l}^G \subseteq \mathfrak{l}^G; \mathfrak{l}^R$*, that is, if* $(\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R$ *and* $(\mathsf{h}, \mathsf{h}_1) \in \mathfrak{l}^G$*, then there exists* h'_1 *such that* $(\mathsf{h}', \mathsf{h}'_1) \in \mathfrak{l}^G$ *and* $(\mathsf{h}_1, \mathsf{h}'_1) \in \mathfrak{l}^R$.
5. *if* $\mathsf{h} : \mathfrak{l}^R$ *and* $\mathsf{h}_1 \supseteq \mathsf{h}$ *then* $(\mathsf{h}_1, \mathsf{h}) \in \mathfrak{l}^R$ *thus* $\mathfrak{l}^R$ *"looks" no further than the currently allocated portion of the heap.*
6. *if* $\mathsf{h} : \mathfrak{l}^R$ *and* $(\mathsf{h} \uplus \mathsf{h}', \mathsf{h}'_1) \in \mathfrak{l}^G$*, then* $\exists \mathsf{h}_1.(\mathsf{h}_1, \mathsf{h}_1) \in \mathfrak{l}^G \wedge \mathsf{h}'_1 = \mathsf{h}_1 \uplus \mathsf{h}'$.

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 relies and guarantees interact well with future extensions of the heap, preserving and being independent of fresh locations.

We write $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$ to denote $(\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R$; $\mathsf{h} \overset{\mathfrak{l}}{\rightarrow} \mathsf{h}_1$ to denote $(\mathsf{h}, \mathsf{h}_1) \in \mathfrak{l}^G$. We also use the notation $supp(\mathfrak{l})$ for $supp(\mathfrak{l}^R) \cup supp(\mathfrak{l}^G)$.

If π is a permutation we define $\pi \cdot \mathfrak{l} = (\pi \cdot \mathfrak{l}^R, \pi \cdot \mathfrak{l}^G)$; it is easy to see that $\pi \cdot \mathfrak{l}$ is an abstract location and $\mathrm{fix}(\pi) \supseteq supp(\mathfrak{l}) \Rightarrow \pi \cdot \mathfrak{l} = \mathfrak{l}$.

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

Definition 4.2 (relocation of abstract locations)**.** *Let* $\mathfrak{l}$ *be an abstract location and* $u : supp(\mathfrak{l}) \to \mathbb{L}$ *be injective. Then* $u \cdot \mathfrak{l}$ *is defined as* $\pi \cdot \mathfrak{l}$ *for some* π *extending* u*. We have* $supp(u \cdot \mathfrak{l}) = u(supp(\mathfrak{l}))$.

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

$$\mathfrak{set}^R_X = \{(\mathsf{h}, \mathsf{h}') \mid \text{If } \mathsf{h} \text{ and } \mathsf{h}' \text{ contain a linked list starting from } X \text{ and these lists contain the same set of integers.}\}$$

$$\mathfrak{set}^G_X = \{(\mathsf{h}, \mathsf{h}_1) \mid \mathsf{h} : \mathfrak{l}^R_X, \mathsf{h}_1 : \mathfrak{l}^R_X \text{ and } \forall \mathsf{l}.\mathsf{l} \in \mathrm{dom}(\mathsf{h}) \setminus F(\mathsf{h}, X) \Rightarrow \mathsf{h}(\mathsf{l}) = \mathsf{h}_1(\mathsf{l})\}$$

where $F(\mathsf{h}, X)$ is the set of the locations in h reached by the linked list starting from X. The rely set, $\mathfrak{set}^R_X$, specifies that two heaps are equivalent when linked lists denote the same set of integers. The guarantee, $\mathfrak{set}^G_X$, 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 Definition 4.1(1): We can relocate one of the two linked lists without compromising relatedness in $\mathfrak{set}^R_X$ 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* $\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n$ *be abstract locations. A heap* h *is separated with respect to* $\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n$*, written* $\mathrm{sep}(\mathsf{h}, \{\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n\})$*, if for all* $1 \le i \le n$. $\mathsf{h} : \mathfrak{l}^R_i$ *and for all* $j \neq i$ *and for any heap* h_1 *such that* $\mathsf{h} \overset{\mathfrak{l}_i}{\rightarrow} \mathsf{h}_1$*, we have* $\mathsf{h} \overset{\mathfrak{l}_j}{\sim} \mathsf{h}_1$ *and, coinductively,* $\mathrm{sep}(\mathsf{h}_1, \{\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n\})$.

Definition 4.4 (Independence)**.** *Abstract locations* $\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n$ *are independent, written* $\perp(\{\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n\})$*, if* $\mathsf{h}_1 : \mathfrak{l}_1, \ldots, \mathsf{h}_n : \mathfrak{l}_n$*, then there exists* h*, where* $\mathrm{sep}(\mathsf{h}, \{\mathfrak{l}_1, \mathfrak{l}_2, \ldots, \mathfrak{l}_n\})$ *and* $\mathsf{h} \overset{\mathfrak{l}_i}{\sim} \mathsf{h}_i$ *for* $1 \le i \le n$.

The definition of independence implies that quotienting by the intersection of the PERs yields a product of the individual quotients. As the separated heap h as defined above is unique up to $\sim$ (with respect to $\mathfrak{l}_1, \ldots, \mathfrak{l}_n$), we use the notation $\mathsf{h}_1 \oplus \mathsf{h}_2 \oplus \cdots \oplus \mathsf{h}_n$ to denote a canonical separated heap.

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

$$P_{set} = \{\mathsf{h} \mid \exists \mathsf{h}_1, \mathsf{h}_2.(\mathsf{h} = \mathsf{h}_1 \uplus \mathsf{h}_2) \wedge ([X \mapsto nil] \xrightarrow{\mathfrak{set}_X} \mathsf{h}_1) \wedge ([Y \mapsto nil] \xrightarrow{\mathfrak{set}_Y} \mathsf{h}_2)\}.$$

This set contains the disjoint unions of any evolution of the linked lists pointed by the concrete locations X and Y.

The following lemma will be helpful for relocating abstract locations and asserting independence.

Lemma 4.5. *If* $supp(\mathfrak{l}_1) \cap supp(\mathfrak{l}_2) = \emptyset$*, then* $\mathfrak{l}_1 \perp \mathfrak{l}_2$.

Proof Let $\mathsf{h}_1 : \mathfrak{l}_1$ and $\mathsf{h}_2 : \mathfrak{l}_2$. Let $L_1 = \mathrm{dom}(\mathsf{h}_1)$ and $L_2 = \mathrm{dom}(\mathsf{h}_2)$. Choose permutations π_1 and π_2 such that $\pi_1 \cdot L_1 \cap \pi_2 \cdot L_2 = \emptyset$ and $\mathrm{fix}(\pi_i) \supseteq supp(\mathfrak{l}^R_i)$. Define the pasted heap $\mathsf{h}_1 \oplus \mathsf{h}_2$ by

$$(\mathsf{h}_1 \oplus \mathsf{h}_2)(\mathsf{l}) = \begin{cases} (\pi_1 \cdot \mathsf{h}_1)(\mathsf{l}), \text{ if } \mathsf{l} \in \pi_1 \cdot L_1 \\ (\pi_2 \cdot \mathsf{h}_2)(\mathsf{l}), \text{ if } \mathsf{l} \in \pi_2 \cdot L_2 \\ \text{undefined otherwise} \end{cases}$$

Thus, in particular, $\mathrm{dom}(\mathsf{h}_1 \oplus \mathsf{h}_2) = \pi_1 \cdot L_1 \uplus \pi_2 \cdot L_2$.

From Def. 4.1(1) we know $(\mathsf{h}_1, \pi_1 \cdot \mathsf{h}_1) \in \mathfrak{l}^R_1$. We also have $(\mathsf{h}_1, \pi_1^{-1} \cdot (\mathsf{h}_1 \oplus \mathsf{h}_2)) \in \mathfrak{l}^R_1$ by Def. 4.1(5), thus $(\pi_1 \cdot \mathsf{h}_1, \mathsf{h}_1 \oplus \mathsf{h}_2) \in \mathfrak{l}^R_1$ so we can conclude by transitivity. The remaining properties are left to the reader. □

Definition 4.6. *If* $\mathfrak{l}_1, \mathfrak{l}_2$ *are independent, we form a joint location* $\mathfrak{l}_1 \otimes \mathfrak{l}_2$ *specified as follows:*

- $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^R = \{(\mathsf{h}, \mathsf{h}') \mid (\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R_1 \cap \mathfrak{l}^R_2$ *and* $\mathrm{sep}(\mathsf{h}, \{\mathfrak{l}_1, \mathfrak{l}_2\})$ *and* $\mathrm{sep}(\mathsf{h}', \{\mathfrak{l}_1, \mathfrak{l}_2\})\}$
- $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^G = (\mathfrak{l}^G_1 \cup \mathfrak{l}^G_2)^*$ *restricted to* $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^R$*;*

Lemma 4.7. *If* $\mathfrak{l}_1$ *and* $\mathfrak{l}_2$ *are independent abstract locations, then the joint location* $\mathfrak{l}_1 \otimes \mathfrak{l}_2$ *is an abstract location.*

Lemma 4.8. *Let* $\mathfrak{l}_1$ *and* $\mathfrak{l}_2$ *be independent abstract locations. Then* $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^G \subseteq \mathfrak{l}^G_1; \mathfrak{l}^G_2; (\mathfrak{l}_1 \otimes \mathfrak{l}_2)^R$.

Lemma 4.9. $\perp(\{\mathfrak{l}_1, \mathfrak{l}_2, \mathfrak{l}_3\})$ *if and only if* $\mathfrak{l}_2 \perp \mathfrak{l}_3$ *and* $\mathfrak{l}_1 \perp (\mathfrak{l}_2 \otimes \mathfrak{l}_3)$.

Definition 4.10. *Let* $\mathbf{1}$ *be the abstract location as follows,* $\mathbf{1}^R = \mathbb{H} \times \mathbb{H}$ *and* $(\mathsf{h}, \mathsf{h}') \in \mathbf{1}^G$ *if and only if* $\mathsf{h} = \mathsf{h}'$*. Notice that* $supp(\mathbf{1}) = \emptyset$.

Lemma 4.11. $\mathfrak{l}_1 \perp \mathfrak{l}_2$ *if and only if and* $\mathfrak{l}_2 \perp \mathfrak{l}_1$*. Moreover,* $\mathbf{1} \otimes \mathfrak{l} = \mathfrak{l}$.

Remark 4.12. *Motivated by the axioms of separation algebras [13] one might conjecture that* $\otimes$ *is cancellative, i.e. that* $\mathfrak{l}_1 \otimes \mathfrak{l} = \mathfrak{l}_2 \otimes \mathfrak{l}$ *implies* $\mathfrak{l}_1 = \mathfrak{l}_2$*. 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 $\mathfrak{l}_X$, parametric with respect to X as follows:

$$\mathfrak{int}^R_X = \{(\mathsf{h}, \mathsf{h}') \mid \mathsf{h}(X) = n \wedge \mathsf{h}'(X) = n' \Rightarrow int(n) = int(n')\}$$

$$\mathfrak{int}^G_X = \{(\mathsf{h}, \mathsf{h}_1) \mid \mathsf{h} : \mathfrak{int}^R_X, \mathsf{h}_1 : \mathfrak{int}^R_X \text{ and } \forall \mathsf{l} \in \mathbb{L}.\mathsf{l} \neq X \Rightarrow \mathsf{h}(\mathsf{l}) = \mathsf{h}_1(\mathsf{l})\}$$

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 $\mathfrak{int}_X$ and $\mathfrak{int}_Y$ are independent, *i.e.*, $\mathfrak{int}_X \perp \mathfrak{int}_Y$. 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 the independence of $\mathfrak{set}_X$ and $\mathfrak{set}_Y$

$$P_{int} = \{\mathsf{h} \mid \exists \mathsf{h}_1, \mathsf{h}_2.(\mathsf{h} = \mathsf{h}_1 \uplus \mathsf{h}_2) \wedge ([X \mapsto 0] \xrightarrow{\mathfrak{int}_X} \mathsf{h}_1) \wedge ([Y \mapsto 0] \xrightarrow{\mathfrak{int}_Y} \mathsf{h}_2)\}.$$

Overlapping references Recall the overlapping references example introduced earlier. Let X be the concrete location encoding a

pair of values. We define abstract locations $\mathfrak{fst}_X$ and $\mathfrak{snd}_X$ by:

$$\begin{aligned}
\mathfrak{fst}_X^R = \quad & \{(\mathsf{h},\mathsf{h}') \mid \mathsf{h}(X) = (a_1,a_2) \wedge \mathsf{h}'(X) = (a_1',a_2') \wedge a_1 = a_1'\} \\
\mathfrak{snd}_X^R = \quad & \{(\mathsf{h},\mathsf{h}') \mid \mathsf{h}(X) = (a_1,a_2) \wedge \mathsf{h}'(X) = (a_1',a_2') \wedge a_2 = a_2'\} \\
\mathfrak{fst}_X^G = \quad & \{(\mathsf{h},\mathsf{h}_1) \mid \mathsf{h} : \mathfrak{fst}_X^R, \mathsf{h}_1 : \mathfrak{fst}_X^R \text{ and} \\
& \qquad \forall \mathsf{l} \in \mathbb{L}.\mathsf{l} \neq X \wedge \mathsf{h}(X) = (a_1,a_2) \wedge \mathsf{h}_1(X) = (a_1',a_2') \Rightarrow \\
& \qquad\qquad \mathsf{h}(\mathsf{l}) = \mathsf{h}_1(\mathsf{l}) \wedge a_2 = a_2'\} \\
\mathfrak{snd}_X^G = \quad & \{(\mathsf{h},\mathsf{h}_1) \mid \mathsf{h} : \mathfrak{snd}_X^R, \mathsf{h}_1 : \mathfrak{snd}_X^R \text{ and} \\
& \qquad \forall \mathsf{l} \in \mathbb{L}.\mathsf{l} \neq X \wedge \mathsf{h}(X) = (a_1,a_2) \wedge \mathsf{h}_1(X) = (a_1',a_2') \Rightarrow \\
& \qquad\qquad \mathsf{h}(\mathsf{l}) = \mathsf{h}_1(\mathsf{l}) \wedge a_1 = a_1'\}
\end{aligned}$$

The rely of $\mathfrak{fst}_X$ (respectively, $\mathfrak{snd}_X$) specifies that two heaps h and h' are equivalent whenever they both store a pair of values in X and the first projections (respectively, second projection) of these pairs are the same. The guarantee of $\mathfrak{fst}_X$ (respectively, $\mathfrak{snd}_X$) specifies that it keeps all other locations alone and does not change the second projection (respectively, first projection) of the pair stored at location X.

By using a coinduction hypothesis for separated heaps similar to the ones used above, it is straightforward to verify that $\mathfrak{fst}_X$ and $\mathfrak{snd}_X$ are independent, although they share the concrete location X.

Lazy Initialization Recall the Lazy Initialization example introduced earlier. Let X be the concrete location used to store the data that is initialized. We define the following abstract location parametric on $i \in \{1,2\}$:

$$\begin{aligned}
\mathfrak{lazy}(i)_X^R = \quad & \{(\mathsf{h},\mathsf{h}') \mid \mathsf{h}(X) = 0 \Rightarrow \mathit{initial}(\mathsf{h}',X) \text{ and} \\
& \qquad \mathsf{h}'(X) = 0 \Rightarrow \mathit{initial}(\mathsf{h},X) \text{ and} \\
& \qquad \mathsf{h}(X) = (a_1,a_2) \wedge \mathsf{h}'(X) = (a_1',a_2') \Rightarrow \mathsf{h}(a_i) = \mathsf{h}'(a_i')\} \\
\mathfrak{lazy}(i)_X^G = \quad & \{(\mathsf{h},\mathsf{h}_1) \mid \mathsf{h} : \mathfrak{l}_X^R, \mathsf{h}_1 : \mathfrak{l}_X^R \text{ and } \mathsf{h}_1(X) = 0 \Rightarrow \mathsf{h}(X) = 0 \text{ and} \\
& \qquad \mathsf{h}(X) = (a_1,a_2) \wedge \mathsf{h}_1(X) = (a_1',a_2') \Rightarrow \\
& \qquad\qquad \forall \mathsf{l}.\mathsf{l} \notin \{X,a_i,a_i'\} \Rightarrow \mathsf{h}(\mathsf{l}) = \mathsf{h}_1(\mathsf{l})\}
\end{aligned}$$

where $\mathit{initial}(\mathsf{h},X)$ is the predicate $\mathsf{h}(X) = 0 \vee (\mathsf{h}(X) = (\mathsf{a}_1,\mathsf{a}_2) \wedge \mathsf{h}(\mathsf{a}_1) = 0 = \mathsf{h}(\mathsf{a}_2))$.

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 *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 category 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 $xu = x'u'$ of morphisms is a pullback if whenever $xv = x'v'$ there is unique t such that $v = ut$ and $v' = u't$. 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 $A(x,y)$ 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 $r_A : \Pi x \in |A|.A(x,x)$ and $s_A : \Pi x,y \in |A|.A(x,y) \to A(y,x)$ 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 $t(-,-)$ means the following: If $(x_i)_i$ and $(y_i)_i$ and $(z_i)_i$ are ascending chains in A with suprema x, y, z, and $p_i \in A(x_i,y_i)$ and $q_i \in A(y_i,z_i)$ are proofs such that $(x_i,y_i,p_i)_i$ and $(y_i,z_i,q_i)_i$ are also ascending chains, with suprema (x,y,p) and (y,z,q), then $(x_i,z_i,t(p_i,q_i))$ is an ascending chain of proofs (by monotonicity of $t(-,-)$) and its supremum is $(x,z,t(p,q))$.

If $p \in A(x,y)$ we may write $p : x \sim y$ or simply $x \sim y$. 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 $f_1 : \Pi x,y \in |A|.A(x,y) \to B(f_0(x),f_0(y))$. Two such pairs $f, g : A \to B$ are* identified *if there exists a continuous function $\mu : \Pi a \in |A|.B(f(a),g(a))$.*

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 $(A \times B)((a,b),(a',b')) = A(a,a') \times B(b,b')$. The ordering is given component-wise.*

The underlying predomain of the function space of $A \Rightarrow B$ of setoids A, B is given by the set of representatives *(f_0,f_1) of morphisms from A to B 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 $p : a \sim a'$. Note that we have $f_1(p) : f_0(a) \sim f_0(a')$ and $\mu(a') : f_0(a') \sim g_0(a')$.

Conversely, if $h : C \times A \to B$ is a morphism represented by (h_0,h_1) we define $\lambda(h) : C \to A \Rightarrow B$ by $(\lambda h)_0(c) = (f_0,f_1)$ where $f_0(a) = h_0(c,a)$ and $f_1(p) = h_1(r_C(c),p)$. We also put $(\lambda h)_1(p : c \sim c')(a) = h_1(p,r_A(a))$. If (h_0',h_1') is another representative of h witnessed by a proof μ then $\lambda c.\lambda a.\mu(c,a)$ 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 $U((a, a', p), (a_1, a_1', p_1)) = A(a, a_1) \times A(a', a_1')$. The ordering is component wise. The morphism $u : U \to A$ is defined by $u_0(a, a', q) = a$ and $u_1(p, p') = p$. The remaining definitions and verifications are straightforward. Note, however, that when proving the uniqueness of fill-ins to establish the pullback property we make use of the fact that pointwise $\sim$-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:*

- *$Y_D = ap\langle id_{D \Rightarrow D}, Y_D\rangle$ (so $Y_D(f)$ is a fixpoint of f).*
- *Suppose that both D, D' have least elements and $u : D \to D'$ satisfy $u_0(\bot) = \bot$. Let $D \Rightarrow D \leftarrow^{g} X \rightarrow^{g'} D' \Rightarrow D'$ be the pullback of $u(-) : D \Rightarrow D \to D \Rightarrow D'$ and $u(-) : D' \Rightarrow D' \to D \Rightarrow D'$. So, intuitively, X contains pairs (f, f') with $f : D \Rightarrow D$, $f' : D' \Rightarrow D'$ and $uf = f'u$. Then $uY_D g = Y_{D'} g'$. Intuitively, this means that if $uf = f'u$ and $u(\bot) = \bot$ then $u(Y_D(f)) = Y_{D'}(f')$.*

Proof Given $f \in D \Rightarrow D$ construct the least fixpoint of f_0 as $\sup_i f_0^i(\bot)$. The passage from f to the least fixpoint being continuous this defines the first part of Y_D, i.e., $(Y_D)_0(f) = \sup_i f_0^i(\bot)$. As for the proof component assume that $p : f \sim f'$. By repeatedly invoking p we then get a family of proofs $p_i : f_0^i(\bot) \sim f_0'^i(\bot)$. 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 $\sup_i p_i : Y_D(f) \sim Y_D(f')$. Thus, we put $(Y_D)_1(p) = \sup_i p_i$ which again is continuous.

It is obvious from the construction as a least fixpoint that the required fixpoint 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 $p_i : uf^i(\bot) \sim f'(u(\bot)) = f'(\bot)$ whose supremum witnesses $u(Y_D(f)) \sim Y_{D'}(f')$. 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 ${}^{x}_{u}\Diamond^{x'}_{u'}$ or $\mathsf{w}\,{}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ (when $\mathsf{w}^{(\prime)} = \mathrm{dom}(x^{(\prime)})$) for a pullback square with morphisms x, u, x', u'. We call the common codomain of x and x' the *apex* of the pullback written $\overline{\mathsf{w}}$, while the common domain of u, u', the *low point* of the square, is written $\underline{\mathsf{w}}$. 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 $\mathsf{w}\,{}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ with apex $\overline{\mathsf{w}}$ is *minimal* if whenever there is another pullback $\mathsf{w}\,{}^{x_1}_{u}\Diamond^{x'_1}_{u'}\mathsf{w}'$ over the same span and with apex $\overline{\mathsf{w}_1}$, then there is a unique morphism $t : \overline{\mathsf{w}} \to \overline{\mathsf{w}_1}$ such that $x_1 = tx$ and $x'_1 = tx'$.

Lemma 6.1. *If a category* **C** *has pullbacks and $\mathsf{w}\,{}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ with apex $\overline{\mathsf{w}}$ is a* minimal *pullback then the morphisms x and x' are* jointly epic*, that is, if $fx = gx$ and $fx' = gx'$ 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 $A \setminus \mathrm{Img}(u) + A'$ with $+$ denoting disjoint union.

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

- *An object (written w and called a world) comprises a finite set of mutually independent abstract locations written $\mathrm{dom}(\mathsf{w})$ and a function tagging each abstract location in w with a region from Regs; we write $\mathsf{w}(\mathsf{r})$ for the set of abstract locations in w tagged with r. We define $supp(\mathsf{w}) = \bigcup_{\mathfrak{l} \in \mathrm{dom}(\mathsf{w})} supp(\mathfrak{l})$.*
- *Let w and w' be two worlds. A morphism from w to w' is given by an injective function $u : supp(\mathsf{w}) \to supp(\mathsf{w}')$ such that for each $\mathfrak{l} \in \mathrm{dom}(\mathsf{w})$ we have $u \cdot \mathfrak{l} \in \mathrm{dom}(\mathsf{w}')$ (cf. Def. 4.2) and moreover, if $\mathfrak{l} \in \mathsf{w}(\mathsf{r})$ then $u \cdot \mathfrak{l} \in \mathsf{w}'(\mathsf{r})$.*
- *We write $\mathcal{I}(\mathsf{w}, \mathsf{w}')$ for the set of morphisms $u : \mathsf{w} \to \mathsf{w}'$ that are set-theoretic inclusions. Note that $u \cdot \mathfrak{l} = \mathfrak{l}$ in this case.*
- *The empty world is denoted $\emptyset$, i.e. $\mathrm{dom}(\emptyset) = \emptyset$.*

For example, a morphism between $\{\mathfrak{int}_X\} \to \{\mathfrak{int}_Y, \mathfrak{set}_Z\}$ (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 $\mathfrak{int}_X$ by $\mathfrak{snd}_Y$ 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 $\otimes$-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 $\mathfrak{l}$ to an abstract region isomorphic to $\mathfrak{l} \otimes \mathfrak{l}'$, which can be elegantly captured by adapting 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* **W** *all morphisms are monomorphisms;* **W** *has pullbacks and all spans in* **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 $u : \mathsf{w} \to \mathsf{w}_1$ and $x : \mathsf{w}_1 \to \mathsf{w}_2$ then we can find w'_1 and $u' : \mathsf{w} \to \mathsf{w}'_1$ and $x' : \mathsf{w}'_1 \to \mathsf{w}_2$ so that $\mathsf{w}_1\,{}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'_1$ 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.

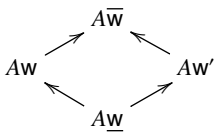
The world w'_1 in the last clause comprises the locations in w and those in w_2 that are not in w_1. Intuitively, if $\mathsf{w} = A$, $\mathsf{w}_1 = A + B$, $\mathsf{w}_2 = A + B + C$ then $\mathsf{w}'_1 = A + C$.

6.1 Setoid-valued functors

A functor A from a category of worlds **W** to the category of setoids comprises as usual for each $\mathsf{w} \in \mathbf{W}$ a setoid $A\mathsf{w}$ and for each $u : \mathsf{w} \to \mathsf{w}'$ a morphism of setoids $Au : A\mathsf{w} \to A\mathsf{w}'$ preserving identities and composition. If $u : \mathsf{w} \to \mathsf{w}'$ and $\mathsf{a} \in A\mathsf{w}$ we may

write $u.\mathsf{a}$ or even $u\mathsf{a}$ for $Au(\mathsf{a})$ and likewise for proofs in $A\mathsf{w}$. Note that $(uv).\mathsf{a} = u.(v.\mathsf{a})$.

Definition 6.4. *We call a functor A* pullback-preserving *(p.p.f.) if for every pullback square $\mathsf{w}\,{}^{x}_{u}\!\diamond^{x'}_{u'}\,\mathsf{w}'$ with apex $\overline{\mathsf{w}}$ and low point $\underline{\mathsf{w}}$ the diagram below is a pullback in the category of setoids.*



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

$$\Pi \mathsf{a} \in A\mathsf{w}.\Pi \mathsf{a}' \in A\mathsf{w}'.A\overline{\mathsf{w}}(x.\mathsf{a}, x'.\mathsf{a}') \to \Sigma \underline{\mathsf{a}} \in A\underline{\mathsf{w}}.A\mathsf{w}(u.\underline{\mathsf{a}}, \mathsf{a}) \times A\mathsf{w}'(u'.\underline{\mathsf{a}}, \mathsf{a}')$$

which amounts to saying that if two values $\mathsf{a} \in A\mathsf{w}$ and $\mathsf{a}' \in A\mathsf{w}'$ are equal in a common world $\overline{\mathsf{w}}$, then this can only be the case because there is a value in the "intersection world" $\underline{\mathsf{w}}$ from which both a and a' arise. The witness $\underline{\mathsf{a}}$ is unique up to $\sim$. 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 $u : \mathsf{w} \to \mathsf{w}'$ and $\mathsf{a}, \mathsf{a}' \in A\mathsf{w}$ then there is a continuous function $A\mathsf{w}'(u.\mathsf{a}, u.\mathsf{a}') \to A\mathsf{w}(\mathsf{a}, \mathsf{a}')$. (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 $A\mathsf{w}'(u.\mathsf{a}, u.\mathsf{a}')$ argument.

Definition 6.6 (Morphism of functors)**.** *If A, B are p.p.fs., 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 $e_0 : \Pi\mathsf{w}.A\mathsf{w} \to B\mathsf{w}$ and*

$$e_1 : \Pi\mathsf{w}.\Pi\mathsf{w}'.\Pi x : \mathsf{w} \to \mathsf{w}'.\Pi a \in A\mathsf{w}.\Pi a' \in A\mathsf{w}'. A\mathsf{w}'(x.\mathsf{a}, \mathsf{a}') \to B\mathsf{w}'(x.e_0(\mathsf{a}), e_0(\mathsf{a}')).$$

Two such morphisms e, e' are equal if there exists a continuous function $\mu : \Pi\mathsf{w}.\Pi a \in A\mathsf{w}.B\mathsf{w}(e(\mathsf{a}), e'(\mathsf{a}))$.

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 $(A \times B)\mathsf{w} = A\mathsf{w} \times B\mathsf{w}$ and $|(A \Rightarrow B)\mathsf{w}|$ contains pairs (f_0, f_1) as follows: $f_0(u) \in |A\mathsf{w}_1 \Rightarrow B\mathsf{w}_1|$ for each w_1 and $u : \mathsf{w} \to \mathsf{w}_1$. If $u : \mathsf{w} \to \mathsf{w}_1$ and $v : \mathsf{w}_1 \to \mathsf{w}_2$ then*

$$f_1(u, v) \in (A\mathsf{w}_1 \Rightarrow B\mathsf{w}_2)([Av \Rightarrow B\mathsf{w}_2]\ f_0(vu), [A\mathsf{w}_1 \Rightarrow Bv]\ f_0(u))$$

where

$$[Av \Rightarrow B\mathsf{w}_2] : (A\mathsf{w}_2 \Rightarrow B\mathsf{w}_2) \to (A\mathsf{w}_1 \Rightarrow B\mathsf{w}_2)$$
$$[A\mathsf{w}_1 \Rightarrow Bv] : (A\mathsf{w}_1 \Rightarrow B\mathsf{w}_1) \to (A\mathsf{w}_1 \Rightarrow B\mathsf{w}_2)$$

are the obvious composition morphisms.

A proof in $(A \Rightarrow B)\mathsf{w}((f_0, f_1), (f_0', f_1'))$ is a function g that for each $u : \mathsf{w} \to \mathsf{w}_1$ yields a proof $g(u) \in (A\mathsf{w}_1 \Rightarrow B\mathsf{w}_1)(f_0(u), f_0'(u))$.

The order on objects and proofs is pointwise as usual.

Moreover, if D is a p.p.f. with the property that $D\mathsf{w}$ has a least element $\bot_{\mathsf{w}}$ for each w and whenever $u : \mathsf{w} \to \mathsf{w}_1$ then $u.\bot_{\mathsf{w}} = \bot_{\mathsf{w}_1}$ then 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 $\mathfrak{S}\mathsf{w}$ is defined by $|\mathfrak{S}\mathsf{w}| = \{\mathsf{h} \in \mathbb{H} \mid \forall \mathfrak{l} \in \mathsf{w}.\mathsf{h} : \mathfrak{l}^R \wedge \mathrm{sep}(\mathsf{h}, \mathrm{dom}(\mathsf{w}))\}$ and $\mathfrak{S}\mathsf{w}(\mathsf{h}, \mathsf{h}') = \{\star\} \iff \forall \mathfrak{l} \in \mathsf{w}.\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$ and $\mathfrak{S}\mathsf{w}(\mathsf{h}, \mathsf{h}') = \emptyset$ otherwise, where $\mathrm{sep}(\mathsf{h}, \mathrm{dom}(\mathsf{w}))$ is the separated heap with respect to the locations in $\mathrm{dom}(\mathsf{w})$ (see Definition 4.3).*

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

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

Lemma 6.9 (Pasting Lemma)**.** *Let $\mathsf{w}\,{}^{x}_{u}\!\diamond^{x'}_{u'}\,\mathsf{w}'$ be a minimal pullback with apex $\overline{\mathsf{w}}$. Let $\mathsf{h} \in \mathfrak{S}\mathsf{w}$ and $\mathsf{h}' \in \mathfrak{S}\mathsf{w}'$ such that $\mathsf{h}.u \sim \mathsf{h}'.u'$. Then there exists a heap $\mathsf{h}_1 \in \mathfrak{S}\overline{\mathsf{w}}$, unique up to $\sim$, such that $\mathsf{h}_1.x \sim \mathsf{h}$ and $\mathsf{h}_1.x' \sim \mathsf{h}'$.*

Corollary 6.10. *Let $u : \mathsf{w} \to \mathsf{w}'$ be a morphism and $\mathsf{h} \in \mathfrak{S}\mathsf{w}$ and $\mathsf{h}' \in \mathfrak{S}\mathsf{w}'$. There exists h_1 unique up to $\sim$ such that $\mathsf{h}_1.u \sim \mathsf{h}$ and $\mathsf{h}_1.u' \sim \mathsf{h}'.u'$ where $\mathsf{w}_0\,{}^{u}_{0}\!\diamond^{u'}_{0}\,\mathsf{w}_1$ is 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 $\mathfrak{S}$ consists of a subset $R\mathsf{w} \subseteq \mathfrak{S}\mathsf{w} \times \mathfrak{S}\mathsf{w}$ for each w, such that if $(\mathsf{h}, \mathsf{h}') \in R\mathsf{w}$ and $u : \mathsf{w}_0 \to \mathsf{w}$ then $(\mathsf{h}.u, \mathsf{h}'.u) \in R\mathsf{w}_0$, and if $p : \mathsf{h} \sim \mathsf{h}_1$ and $p' : \mathsf{h}' \sim \mathsf{h}_1'$ then $(\mathsf{h}_1, \mathsf{h}_1') \in R\mathsf{w}$, as well. Given a minimal pullback $\mathsf{w}\,{}^{x}_{u}\!\diamond^{x'}_{u'}\,\mathsf{w}'$ and $\mathsf{h}, \mathsf{h}' \in \mathfrak{S}\overline{\mathsf{w}}$ such that $(h.x, h'.x) \in R\mathsf{w}$ and $(h.x', h'.x') \in R\mathsf{w}'$ then $(h, h') \in R\overline{\mathsf{w}}$.*

The elementary effects track reading, writing, and allocating at the level of regions and are as follows: wr_{r} (writing within region r), rd_{r} (reading from within region r), and al_{r} (allocating within region r). Each elementary effect ϵ is associated with a set $\mathcal{R}(\epsilon)$ of relations on $\mathfrak{S}$ as follows:

$$R \in \mathcal{R}(rd_{\mathsf{r}}) \iff (\mathsf{h}, \mathsf{h}') \in R\mathsf{w} \Rightarrow \forall \mathfrak{l} \in \mathsf{w}(\mathsf{r}).\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$$

$$R \in \mathcal{R}(wr_{\mathsf{r}}) \iff (\mathsf{h}, \mathsf{h}') \in R\mathsf{w} \Rightarrow \forall \mathfrak{l} \in \mathsf{w}(\mathsf{r}).\forall \mathsf{h}_1, \mathsf{h}_1'.\mathsf{h}_1 \overset{\mathfrak{l}}{\sim} \mathsf{h}_1' \Rightarrow \mathsf{h} \overset{\mathfrak{l}}{\to} \mathsf{h}_1 \wedge \mathsf{h}' \overset{\mathfrak{l}}{\to} \mathsf{h}_1' \Rightarrow (\mathsf{h}_1, \mathsf{h}_1') \in R\mathsf{w}$$

$$R \in \mathcal{R}(al_{\mathsf{r}}) \iff (\mathsf{h}, \mathsf{h}') \in R\mathsf{w} \Rightarrow \forall \mathsf{w}_1.\forall u \in \mathcal{I}(\mathsf{w}, \mathsf{w}_1).(\mathsf{w}_1 \setminus \mathsf{w}) \subseteq \mathsf{w}_1(\mathsf{r}) \Rightarrow \forall \mathsf{h}_1, \mathsf{h}_1' \in \mathfrak{S}\mathsf{w}_1.[\mathsf{h}_1.u \sim \mathsf{h} \wedge \mathsf{h}_1'.u \sim \mathsf{h}' \wedge (\mathsf{h}_1, \mathsf{h}_1') \in \textstyle\bigcap_{\mathfrak{l} \in \mathsf{w}_1 \setminus \mathsf{w}} \mathfrak{l}^R] \Rightarrow (\mathsf{h}_1, \mathsf{h}_1') \in R\mathsf{w}_1$$

Thus, $\mathcal{R}(rd_{\mathsf{r}})$ is the set of relations R for which R-related heaps contain "equal" (in the sense of $\mathfrak{l}^R$) values for all abstract locations in region r; a relation $R \in \mathcal{R}(wr_{\mathsf{r}})$ is oblivious to (preserved by) "equal" writes to any abstract location in r; and a relation $R \in \mathcal{R}(al_{\mathsf{r}})$ 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 $(-)^R$, and that nothing else is changed.

Finally, $\mathcal{R}(\varepsilon)$ for a set of effects ε is defined as follows:

$$\mathcal{R}(\varepsilon) = \bigcap_{\epsilon \in \varepsilon} \mathcal{R}(\epsilon).$$

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

Definition 6.12 (Monad)**.** *Let A be a p.p.f. and ε an effect. A p.p.f. $T_\varepsilon A$ is defined as follows:*

- *(Objects) Elements of $(T_\varepsilon A)\mathsf{w}$ are pairs $(\mathsf{c}_0, \mathsf{c}_1)$ of partial functions where*

$$\mathsf{c}_0 : \quad \Subset \mathsf{w} \rightharpoonup \Sigma \mathsf{w}_1 . \mathcal{I}(\mathsf{w}, \mathsf{w}_1) \times \Subset \mathsf{w}_1 \times A\mathsf{w}_1$$

and c_1 is as follows. If $R \in \mathcal{R}(\varepsilon)$ and $(\mathsf{h}, \mathsf{h}') \in R\mathsf{w}$ then $\mathsf{c}_1(R, \mathsf{h}, \mathsf{h}')$ either is undefined and $\mathsf{c}_0(\mathsf{h})$ and $\mathsf{c}_0(\mathsf{h}')$ are both undefined or else $c_1(R, \mathsf{h}, \mathsf{h}')$ is defined and then $\mathsf{c}_0(\mathsf{h})$ and $\mathsf{c}'_0(\mathsf{h}')$ are both defined, say $\mathsf{c}_0(\mathsf{h}) = (\mathsf{w}_1, u, \mathsf{h}_1, \mathsf{a})$ and $\mathsf{c}_0(\mathsf{h}') = (\mathsf{w}'_1, u', \mathsf{h}'_1, \mathsf{a}')$. In this case, $\mathsf{c}_1(R, \mathsf{h}, \mathsf{h}')$ returns a pair $({}^{x}_{v}\Diamond^{x'}_{v'}, p)$ where $\mathsf{w}_1 {}^{x}_{v}\Diamond^{x'}_{v'} \mathsf{w}'_1$ such that $xu = x'u'$. Furthermore, $p \in A\overline{\mathsf{w}}(x.\mathsf{a}, x'.\mathsf{a}')$ and, finally, $(\mathsf{h}_1.v, \mathsf{h}'_1.v') \in R\underline{\mathsf{w}}$ where $\underline{\mathsf{w}}$ and $\overline{\mathsf{w}}$ are low point and apex of ${}^{x}_{v}\Diamond^{x'}_{v'}$.

- *(Proofs) Proofs only look at the $(-)_0$ components. Let $\mathsf{c} = (\mathsf{c}_0, \_) \in T_\varepsilon A\mathsf{w}$ and $\mathsf{c}' = (\mathsf{c}'_0, \_) \in T_\varepsilon A\mathsf{w}$, then a proof in $(T_\varepsilon A)\mathsf{w}(\mathsf{c}, \mathsf{c}')$ is a partial (continuous) function μ, such that for a given $\mathsf{h} \in \Subset \mathsf{w}$:*
  1. *if $\mu(\mathsf{h})$ is undefined, then $\mathsf{c}_0(\mathsf{h})$ and $\mathsf{c}'_0(\mathsf{h})$ are both undefined;*
  2. *if $\mu(\mathsf{h})$ is defined, then $\mathsf{c}_0(\mathsf{h}) = (\mathsf{w}_1, u, \mathsf{h}_1, \mathsf{a})$ and $\mathsf{c}'_0(\mathsf{h}) = (\mathsf{w}'_1, u', \mathsf{h}'_1, \mathsf{a}')$ are both defined. In this case, $\mu(\mathsf{h})$ returns a tuple $({}^{x}_{v}\Diamond^{x'}_{v'}, q)$ satisfying $xu = x'u'$, $q \in A\overline{\mathsf{w}}(x.\mathsf{a}, x'.\mathsf{a}')$ and $\mathsf{h}_1.v \sim \mathsf{h}_1.v'$ in $\Subset \underline{\mathsf{w}}$, with $\overline{\mathsf{w}}$ and $\underline{\mathsf{w}}$ apex and low point of ${}^{x}_{v}\Diamond^{x'}_{v'}$.*

- *(Order) The order between computations is given point-wise: $(\mathsf{c}_0, \mathsf{c}_1) \leq (\mathsf{c}'_0, \mathsf{c}'_1)$ if and only if for any $\mathsf{h} \in \Subset \mathsf{w}$ we have $\mathsf{c}_0(\mathsf{h}) \leq \mathsf{c}'_0(\mathsf{h})$, which is defined as follows: for all $\mathsf{h} \in \Subset \mathsf{w}$ if $\mathsf{c}_0(\mathsf{h})$ is defined so is $\mathsf{c}'_0(\mathsf{h})$. Moreover, if both $\mathsf{c}_0(\mathsf{h})$ and $\mathsf{c}'_0(\mathsf{h})$ are defined as $\mathsf{c}_0(\mathsf{h}) = (\mathsf{w}_1, u_1, \mathsf{h}_1, \mathsf{a}_1)$ and $\mathsf{c}'_0(\mathsf{h}) = (\mathsf{w}'_1, u'_1, \mathsf{h}'_1, \mathsf{a}'_1)$, then $\mathsf{w}_1 = \mathsf{w}'_1$, $u_1 = u'_1$, $\mathsf{h}_1 = \mathsf{h}'_1$ and $\mathsf{a}_1 \leq \mathsf{a}'_1$. Finally, if $\mathsf{c}_1(R, \mathsf{h}, \mathsf{h}')$ and $\mathsf{c}'_1(R, \mathsf{h}, \mathsf{h}')$ are defined returning $({}^{x_1}_{v_1}\Diamond^{x'_1}_{v'_1}, q)$ and $({}^{x_2}_{v_2}\Diamond^{x'_2}_{v'_2}, q')$, then ${}^{x_1}_{v_1}\Diamond^{x'_1}_{v'_1} = {}^{x_2}_{v_2}\Diamond^{x'_2}_{v'_2}$ and $q \leq q'$.*[1]

- *(Morphism part) Let $x : \mathsf{w} \to \mathsf{w}'$ be a morphism and $\mathsf{c} = (\mathsf{c}_0, \mathsf{c}_1) \in T_\varepsilon A\mathsf{w}$. We are going to describe the first component of $(\mathsf{c}'_0, \_) = T_\varepsilon Ax(\mathsf{c})$. Given $\mathsf{h} \in \Subset \mathsf{w}'$ write $\mathsf{c}_0(\mathsf{h}.x) = (\mathsf{w}_1, u, \mathsf{h}_1, \mathsf{a})$. Should this application be undefined so is $\mathsf{c}'_0(\mathsf{h})$. Now, using Proposition 6.3, complete u, x to a minimal pullback square ${}^{x'}_{u}\Diamond^{u'}_{x}$ with $u' : \mathsf{w}' \to \mathsf{w}'_1$ where w'_1 is the apex of this pullback.*

  Now, obtain $\mathsf{h}' \in \Subset \mathsf{w}'$ from h by overwriting with $\mathsf{h}_1.u$ using Corollary 6.10. Then paste h_1 and h' (note that $\mathsf{h}_1.u \sim \mathsf{h}'.u$ by construction) using Lemma 6.9 to yield the desired heap $\mathsf{h}'_1 \in \Subset \mathsf{w}'_1$. We then put $\mathsf{c}_0(\mathsf{h}) = (\mathsf{w}'_1, u', \mathsf{h}'_1, x'.\mathsf{a})$.

  We omit the remaining components and verifications.

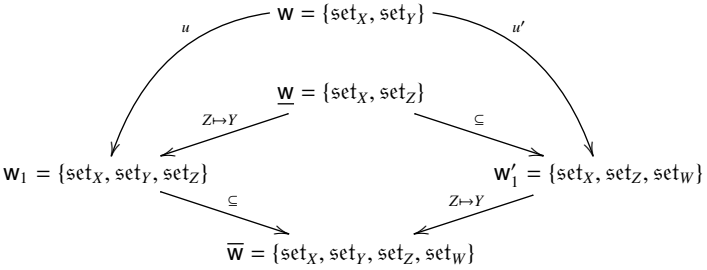
The proof of the following is tedious but straightforward and will be given in an extended version of this paper.

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

This theorem means in particular, that each T_ε is a monad on the category of p.p.f. *i.e.*, for each A there are natural transformation $\eta_A : A \to T_\varepsilon A$ and $\mu_A : T_\varepsilon T_\varepsilon A \to T_\varepsilon A$ subject to the usual laws and whenever $\varepsilon_1 \subseteq \varepsilon_2$ then there is a morphism $T_{\varepsilon_1} A \to T_{\varepsilon_2} A$ 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 $\mathsf{h}(X) \mapsto [1, 2, 3]$ and $\mathsf{h}(Y) \mapsto [3, 2, 1]$, that is, h points to the lists $[1, 2, 3]$ and $[3, 2, 1]$ from concrete locations X and Y, respectively. Moreover, let $\mathsf{c}(\mathsf{h}) = (\mathsf{w}_1, u, \mathsf{h}_1, (3, Y))$ and $\mathsf{c}'(\mathsf{h}) = (\mathsf{w}'_1, u', \mathsf{h}'_1, (3, Z))$, where $\mathsf{h}_1(X) \mapsto [1, 2, 3, 4]$; $\mathsf{h}_1(Y) \mapsto [3, 2, 1, 0]$; and $\mathsf{h}_1(Z) = [4, 2]$, while $\mathsf{h}'_1(X) \mapsto [4, 3, 2, 1]$; $\mathsf{h}'_1(Z) \mapsto [3, 2, 1, 3, 2, 1, 0]$; and $\mathsf{h}'_1(W) = [0, 0]$. 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.*, $Z \mapsto Y$, denotes the corresponding renaming.



Here, it is easy to check that h_1 and h'_1 are equal when taken to the world $\underline{\mathsf{w}}$, as the concrete locations X in h_1 and h'_1 and Y in h_1 and Z in h'_1 point to lists containing the same set of elements: $\{1, 2, 3, 4\}$ and $\{0, 1, 2, 3\}$, respectively. Notice that the abstract locations $\mathrm{set}_Z \in \mathsf{w}_1$ and $\mathrm{set}_W \in \mathsf{w}'_1$ are not taken into account. Moreover, the value $(3, Y)$ is also equivalent to $(3, Z)$ when taken to the world $\overline{\mathsf{w}}$, as the morphism $\mathsf{w}'_1 \to \overline{\mathsf{w}}$ renames Z to Y, while the morphism $\mathsf{w}_1 \to \overline{\mathsf{w}}$ is an inclusion.

We see here that the use of setoids rather than plain sets or predomains is crucial. The proofs in $T_\varepsilon A$ are clearly relevant as they explain in what way the abstract locations correspond to each other. Likewise, the elements of $|T_\varepsilon A|$ 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 $T_\varepsilon A$ 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 $[\![\tau_1 \xrightarrow{\varepsilon} \tau_2]\!] = [\![\tau_1]\!] \Rightarrow T_\varepsilon \tau_2$. We also interpret a typing context Γ as the cartesian product over its bindings: $[\![\Gamma]\!] = \prod_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]$.

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

Definition 7.3 (axiom interpretation)**.** *An axiom interpretation consists of an element $x_{(v,\tau)} \in |[\![\tau]\!]\emptyset|$ (empty world) for each type axiom (v, τ) and of an element $p_{(v,v',\tau)} \in [\![\tau]\!]\emptyset(x_{(v,\tau)}, x_{(v',\tau)})$ for each equality axiom (v, v', τ).*

Given an axiom interpretation (and a type interpretation) and a well-typed term $\Gamma \vdash e : \tau \;\&\; \varepsilon$ we define a morphism

$$[\![\Gamma \vdash e : \tau \;\&\; \varepsilon]\!] : [\![\Gamma]\!] \to T_\varepsilon [\![\tau]\!]$$

using the fact that all the typing rules are reflected in the semantics: cartesian closure takes care of application and abstraction and

variable management. The fixpoint morphism Y_D takes care of recursive definitions; the indexed monadicity of T_ε takes care of the rules associated with "let" and effect propagation. Lemma 7.2, finally, takes care of the masking rule. If $\Gamma \vdash v : \tau$ because of the axiom (v, τ) then we put $[\![\Gamma \vdash v : \tau]\!](\gamma) = x_{(v,\tau)}$ (constant morphism).

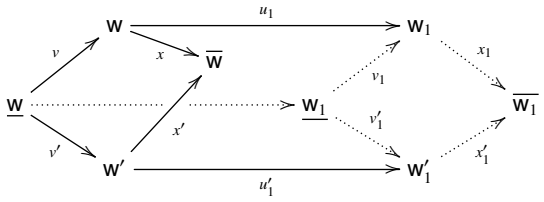
Effects Given a set of effects ε, we write $\mathrm{rds}(\varepsilon) = \{\mathsf{r} \mid rd_{\mathsf{r}} \in \varepsilon\}$, $\mathrm{wrs}(\varepsilon) = \{\mathsf{r} \mid wr_{\mathsf{r}} \in \varepsilon\}$, $\mathrm{als}(\varepsilon) = \{\mathsf{r} \mid al_{\mathsf{r}} \in \varepsilon\}$ and $\mathrm{regs}(\varepsilon) = \mathrm{rds}(\varepsilon) \cup \mathrm{wrs}(\varepsilon) \cup \mathrm{als}(\varepsilon)$. Moreover, the set $nwrs(\varepsilon) = \mathrm{regs}(\varepsilon) \setminus \mathrm{wrs}(\varepsilon)$. We also introduce the following piece of notation for $\mathsf{h}, \mathsf{h}' \in \mathfrak{S}\mathsf{w}$:

$$\mathsf{h} \sim_{\mathrm{rds}(\varepsilon,\mathsf{w})} \mathsf{h}' \iff \forall \mathsf{l} \in \mathsf{w}(\mathrm{rds}(\varepsilon)).\mathsf{h} \overset{\mathsf{l}}{\sim} \mathsf{h}'$$

$$\mathsf{h} \sim_{nwrs(\varepsilon,\mathsf{w})} \mathsf{h}' \iff \forall \mathsf{l} \in \mathsf{w}(nwrs(\varepsilon)).\mathsf{h} \overset{\mathsf{l}}{\sim} \mathsf{h}'$$

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

Lemma 7.4. *Let $\Gamma \vdash e : \tau \;\&\; \varepsilon$. Let $\mathsf{w} {}^{x}_{v}\!\Diamond^{x'}_{v'} \mathsf{w}'$ be a pullback square, $\gamma \in [\![\Gamma]\!]\mathsf{w}, \gamma' \in [\![\Gamma]\!]\mathsf{w}'$ be contexts, such that $\mu_\gamma : x.\gamma \sim x.\gamma$ and $\mathsf{c} = [\![\Gamma \vdash e : \tau \;\&\; \varepsilon]\!]\mathsf{w}(\gamma) \in T_\varepsilon A\mathsf{w}$ and $\mathsf{c}' = [\![\Gamma \vdash e : \tau \;\&\; \varepsilon]\!]\mathsf{w}'(\gamma') \in T_\varepsilon A\mathsf{w}'$ be computations, such that $\mu : x.\mathsf{c} \sim x'.\mathsf{c}'$. Let $\mathsf{h} \in \mathfrak{S}\mathsf{w}, \mathsf{h}' \in \mathfrak{S}\mathsf{w}'$ be heaps, such that $\mathsf{h}.v \sim_{\mathrm{rds}(\varepsilon,\underline{\mathsf{w}})} \mathsf{h}'.v'$. Then $\mathsf{c}(\mathsf{h})$ and $\mathsf{c}'(\mathsf{h}')$ coterminate. Moreover, if they do terminate and $\mathsf{c}(\mathsf{h}) = (\mathsf{w}_1, u_1, \mathsf{h}_1, \mathsf{a}_1)$ and $\mathsf{c}'(\mathsf{h}') = (\mathsf{w}'_1, u'_1, \mathsf{h}'_1, \mathsf{a}'_1)$, then the following diagram exists, in particular, the dashed arrows, where the following is satisfied*



1. $x_1.\mathsf{a} \sim x'_1.\mathsf{a}'$, $\mathsf{w} \cong \underline{\mathsf{w}} \otimes \mathsf{q}$, $\mathsf{w}' \cong \underline{\mathsf{w}} \otimes \mathsf{q}'$, $\mathsf{w}_1 \cong \underline{\mathsf{w}} \otimes \mathsf{q} \otimes \mathsf{q}_1$, $\mathsf{w}'_1 \cong \underline{\mathsf{w}} \otimes \mathsf{q}' \otimes \mathsf{q}'_1$, *for some* q_1 *and* q'_1*;*
2. *for all* $\mathsf{l} \in \underline{\mathsf{w}}$*, we have either:* $\mathsf{h}.v \overset{\mathsf{l}}{\sim} \mathsf{h}_1.u_1.v$ *and* $\mathsf{h}'.v' \overset{\mathsf{l}}{\sim} \mathsf{h}'_1.u'_1.v'$ *(remain equivalent) or* $\mathsf{h}_1.u_1.v_1 \overset{\mathsf{l}}{\sim} \mathsf{h}'_1.u'_1.v'$ *(equally modified);*
3. *if* $\mathsf{l} \in \underline{\mathsf{w}}(nwrs(\varepsilon))$*, then* $\mathsf{h}.v \overset{\mathsf{l}}{\sim} \mathsf{h}_1.u_1.v$ *and* $\mathsf{h}'.v' \overset{\mathsf{l}}{\sim} \mathsf{h}'_1.u'_1.v'$*.*
4. *if* $\mathrm{als}(\varepsilon) = \emptyset$*, then there exists a morphism* $\mathsf{c}_\star \in [\![\Gamma]\!] \to T_\varepsilon[\![\tau]\!]$*, such that* $\mathsf{c}_\star \sim \mathsf{c}$ *and if* $\mathsf{c}_\star(\mathsf{w})(\gamma)\mathsf{h} = (\mathsf{w}_\star, u_\star, \mathsf{h}_\star, \mathsf{a}_\star)$ *then* $\mathsf{w}_\star \cong \mathsf{w}$ *(no world extension is needed);*
5. $\underline{\mathsf{w}}\mathsf{w}\underline{\mathsf{w}_1}\mathsf{w}_1$ *and* $\underline{\mathsf{w}}\mathsf{w}'\underline{\mathsf{w}_1}\mathsf{w}'_1$ *are pullbacks.*

Proof The proof that the values are equal in $\overline{\mathsf{w}_1}$ follows directly from the definition of computations and effects.

For the item 2, we use the following relation R defined as follows for all worlds w_1, such that $u : \underline{\mathsf{w}} \to \mathsf{w}_1$:

$$\{(\mathsf{h}_1, \mathsf{h}'_1) \mid \mathsf{h}_1.u \sim_{\mathrm{rds}(\varepsilon,\underline{\mathsf{w}})} \mathsf{h}'_1.u \wedge \forall \mathsf{l} \in \underline{\mathsf{w}}.(\mathsf{h}_1.u \overset{\mathsf{l}}{\sim} \mathsf{h} \wedge \mathsf{h}'_1.u \overset{\mathsf{l}}{\sim} \mathsf{h}') \vee \mathsf{h}_1.u \overset{\mathsf{l}}{\sim} \mathsf{h}'_1.u\}$$

Otherwise, for the worlds w_2 not reachable from $\underline{\mathsf{w}}$, the relation $R\mathsf{w}_2$ is the trivial set, that is, all heaps are related to each other. Notice that $R \in \mathcal{R}(\varepsilon)$ and it is contravariant. The claim then follows directly by using the morphism from $\underline{\mathsf{w}} \to \underline{\mathsf{w}_1}$.

The proof of item 3 follows in a similar fashion, but we use the following relation:

$$\{(\mathsf{h}_1, \mathsf{h}'_1) \mid \mathsf{h}_1.u \sim_{\mathrm{rds}(\varepsilon,\underline{\mathsf{w}})} \mathsf{h}'_1.u \wedge \mathsf{h}_1.u \sim_{nwrs(\varepsilon,\underline{\mathsf{w}})} \mathsf{h}\}$$

And we use a similar relation for showing that h' and $\mathsf{h}'_1.u'$ agree on the not written locations $\underline{\mathsf{w}}(nwrs(\varepsilon))$.

For item 4, first, we show that $\mathsf{w}(\mathsf{r})$ and $\underline{\mathsf{w}}(\mathsf{r})$ are equal up to renaming for all regions $\mathsf{r} \notin \mathrm{als}(\mathsf{r})$ by using the following relation $R\mathsf{q}$ for some world q:

$$\{(\mathsf{h}_1, \mathsf{h}'_1) \mid \mathsf{h}_1 \sim \mathsf{h}'_1 \wedge \forall \mathsf{r} \notin \mathrm{als}(\varepsilon).\#_{\mathsf{r}}(\mathsf{q}) \leq \#_{\mathsf{r}}(\mathsf{w})\}$$

where $\#_{\mathsf{r}}$ denotes the number of abstract locations colored with r. Clearly, $R \in \mathcal{R}(\varepsilon)$ as ε does not contain any allocation effects. Thus, the number of locations in $\mathsf{w}_\star$ is the same as the locations in w, by using the pullback $\mathsf{w}_1 {}^{\mathsf{l}}_{v}\!\Diamond^{x}_{1} \mathsf{w}_\star$. This gives us one direction, while the other direction is obtained by the universal property of pullbacks. Given this property, one can easily construct the function $\mathsf{c}_\star$. □

Theorem 7.5 (Equational soundness)**.** *Assume an ambient interpretation of types and axioms. If $\Gamma \vdash e = e' : \tau \;\&\; \varepsilon$ then the morphisms $[\![\Gamma \vdash e : \tau \;\&\; \varepsilon]\!]$ and $[\![\Gamma \vdash e' : \tau \;\&\; \varepsilon]\!]$ 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 $\Gamma \vdash v = v' : \tau$ using the axiom (v, v', τ) then we have $[\![\Gamma \vdash v : \tau]\!](\gamma) = x_{(v,\tau)}$ and $[\![\Gamma \vdash v' : \tau]\!](\gamma) = x_{(v',\tau)}$ and the constant function $\lambda\gamma.p_{(v,v',\tau)}$ 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 τ, effect ε, and world w we define admissible relations $\Vdash^{\tau}_{\mathbf{W}} \subseteq \mathbb{V} \times |[\![\tau]\!]\mathbf{W}|$ and $\Vdash^{\tau\&\varepsilon}_{\mathbf{W}} \subseteq \mathbb{C} \times |T_\varepsilon[\![\tau]\!]\mathbf{W}|$ as follows:*

- *If $c \in \mathbb{C}$ then $c \Vdash^{\tau\&\varepsilon}_{\mathsf{w}} (\mathsf{c}_0, \mathsf{c}_1)$ means that for each $\mathsf{h} \in \mathfrak{S}\mathsf{w}$ one has $c(\mathsf{h})$ defined iff $\mathsf{c}_0(\mathsf{h})$ defined and if $c(\mathsf{h}) = (\mathsf{h}_1, a)$ then $\mathsf{c}_0(\mathsf{h}) = (\mathsf{w}_1, u, \mathsf{h}_1, \mathsf{a})$ (same h_1) and $\mathsf{h}_1 \in \mathfrak{S}\mathsf{w}_1$ and $a \Vdash^{\tau}_{\mathsf{w}_1} \mathsf{a}$.*
- *If $fun(f) \in \mathbb{V}$ and $(f_0, f_1) \in |[\![\tau_1 \overset{\varepsilon}{\to} \tau_2]\!]\mathsf{w}|$ then $f \Vdash^{\tau_1 \overset{\varepsilon}{\to} \tau_2}_{\mathsf{w}} (f_0, f_1)$ iff whenever $u : \mathsf{w} \to \mathsf{w}_1$ is an inclusion and $a \Vdash^{\tau_1}_{\mathsf{w}_1} x$ then $f(a) \Vdash^{\tau_2\&\varepsilon}_{\mathsf{w}_1} f_0(x)$.*
- *$\eta \Vdash^{\Gamma}_{\mathsf{w}} \gamma \iff \eta(x) \Vdash^{\Gamma(x)}_{\mathsf{w}} \gamma(x)$ for all $x \in \mathrm{dom}(\Gamma)$.*

Lemma 8.2. *The $\Vdash$ relations are indeed admissible and are stable under inclusion; that is whenever $v \Vdash^{X}_{\mathsf{w}} x$ and $u : \mathsf{w} \to \mathsf{w}'$ is an inclusion then $v \Vdash^{X}_{\mathsf{w}_1} u.x$ holds.*

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 (v, τ) one has $v \Vdash^{\tau}_{\emptyset} x_{(v,\tau)}$. Whenever $\Gamma \vdash e : \tau \;\&\; \varepsilon$ and $\eta \Vdash^{\Gamma}_{\mathsf{w}} \gamma$ then $[\![e]\!]\eta \Vdash^{\tau\&\varepsilon}_{\mathsf{w}} [\![\Gamma \vdash e : \tau]\!]_{\mathsf{w}}(\gamma)$.*

Definition 8.4 (Observational equivalence)**.** *Two value expressions v, v' where $\vdash v : \tau$ and $\vdash v' : \tau$ are observationally equivalent at type τ if for all f such that $\vdash f : \tau \overset{\varepsilon}{\to} \mathtt{unit}$ ("observations") it is the case that $[\![f\ v]\!]\emptyset$ is defined iff $[\![f\ v']\!]\emptyset$ is defined. ($\emptyset$ stands for empty heap and environment).*

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 λ-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 $\vdash v = v' : \tau$ then v and v' are observationally equivalent.*

Proof If f is an observation then by the congruence rules we get $\vdash f\ v = f\ v' : \tau$ so, by Theorem 7.5 we find $[\![\vdash f\ v]\!] = [\![\vdash f\ v']\!]$, thus in particular $[\![\vdash f\ v]\!]_\emptyset\emptyset$ (empty world, empty heap) is

defined iff $[\![\vdash f\ v']\!]_\emptyset\emptyset$ is defined. Now, if $[\![f\ v]\!]\emptyset$ is defined then by Theorem 8.3 and the definition of $\Vdash$ we can conclude that $[\![\vdash f\ v]\!]_\emptyset\emptyset$ is defined so, using Theorem 8.3 again we get that $[\![f\ v']\!]\emptyset$ is defined. □

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_{or} and type τ_{or} from Section 2 (two overlapping references). In order to justify the axiom (v_{or}, τ_{or}) we have to construct a semantic object $x_{or} \in [\![\tau_{or}]\!]$ such that $v_{or} \Vdash^{\tau_{or}}_{\emptyset} x_{or}$ which we will now do.

Given world w we put $x_{or}(\mathsf{w})(\star) = (\mathsf{c}_0, \mathsf{c}_1)$, where c_0 and c_1 are defined as follows ($\star \in [\![\mathtt{unit}]\!]\mathsf{w}$):

$$\mathsf{c}_0(\mathsf{h}) = (u : \mathsf{w} \to \mathsf{w}_1, \mathsf{h}_1, (\mathsf{g}_1, \mathsf{g}_2, \mathsf{s}_1, \mathsf{s}_2))$$

where $new(\mathsf{h}, 0) = (X, \mathsf{h}_1)$, that is, h_1 is obtained from h by allocating a concrete location X; w_1 is the world obtained by extending w with the abstract locations $\mathfrak{fst}_X$ and $\mathfrak{snd}_X$ defined in Section 4.1, where the former is marked with region r1 and the latter with region r2; finally, $(\mathsf{g}_1, \mathsf{g}_2, \mathsf{s}_1, \mathsf{s}_2)$ are the corresponding semantic objects that get and set values to the projections of pairs stored in X, *e.g.*, $\mathsf{g}_1 \in [\![(unit \xrightarrow{rd_{r1}} int)]\!]\mathsf{w}_1$ is the semantic counter part of the function getting the first projection of the pair stored in X, defined in a similar way as we do for x_{or}.

Note that $v_{or} \Vdash^{\tau_{or}}_{\emptyset} x_{or}$ in particular requires that the new heap returned by c_0 agrees with the one returned by v_{or} and, more generally, we are forced to follow the computational behavior of v_{or} in x_{or}. 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 $R \in \mathcal{R}(\{al_{r1}, al_{r2}\})$ is a heap relation and two heaps $\mathsf{h}, \mathsf{h}' \in \Im\mathsf{w}$ that are related, *i.e.*, $(\mathsf{h}, \mathsf{h}') \in R$. Moreover, assume that $\mathsf{c}_0(\mathsf{h}) = (u : \mathsf{w} \to \mathsf{w} \otimes \mathfrak{fst}_X \otimes \mathfrak{snd}_X, \mathsf{h}_1, (\mathsf{g}_1, \mathsf{g}_2, \mathsf{s}_1, \mathsf{s}_2))$ and $\mathsf{c}_0(\mathsf{h}') = (u : \mathsf{w} \to \mathsf{w} \otimes \mathfrak{fst}_{X'} \otimes \mathfrak{snd}_{X'}, \mathsf{h}'_1, (\mathsf{g}'_1, \mathsf{g}'_2, \mathsf{s}'_1, \mathsf{s}'_2))$, where the former allocates X, while the latter allocates X'. Note that everything terminates in this example. Now $\mathsf{c}_1(R, \mathsf{h}, \mathsf{h}')$ returns the pullback $\mathsf{w}_1 {}^{X \mapsto X'}_{1}\!\diamond^{1}_{X \mapsto X'} \mathsf{w}'_1$, 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 $(\mathsf{g}_1, \mathsf{g}_2, \mathsf{s}_1, \mathsf{s}_2)$ and $(\mathsf{g}'_1, \mathsf{g}'_2, \mathsf{s}'_1, \mathsf{s}'_2)$ 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" commute.

Set factory Let w be a world and $\mathsf{h} \in \Im\mathsf{w}$. Suppose that h_1 arises from h by allocating a fresh set data structure, *e.g.*, a linked list, with entry point X. Let $\mathfrak{set}_X$ be the abstract location describing this fresh data structure.

Now for any chosen region r we can add $\mathfrak{set}_X$ to r to yield a new world w_1. The function $setfactory_0(\mathsf{w})(\mathsf{h})$ then returns w_1 and a tuple of semantic functions for reading, membership, removal of which we only sketch reading here: If $u : \mathsf{w}_1 \to \mathsf{w}_2$ and $\mathsf{h}_1 \in \Im\mathsf{w}_1$ and $i \in \mathbb{Z}$ then the reading function looks up i in the data structure starting at the entry points X in h_1 (note that $\mathsf{h}_1 \in \Im\mathsf{w}$ 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.g.*, removal of duplicates, might have occurred, but no world extension is needed and $\mathsf{h}_1 \sim \mathsf{h}_2$ holds. This together with the fact that the outcome only depends on the $\mathfrak{set}^R_X$ 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 $\mathfrak{l}$ whose specification looks only at the two newly allocated concrete locations, say $\mathsf{l}_x, \mathsf{l}_y$, 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 $\mathfrak{l}$ rely specifies that two heaps are equivalent when the values stored in l_x and l_y are the same: $(\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R \iff \mathsf{h}(\mathsf{l}_x), \mathsf{h}'(\mathsf{l}_x) \wedge \mathsf{h}(\mathsf{l}_y), \mathsf{h}'(\mathsf{l}_y)$. 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:

$e_1 = \mathtt{let}\, x \Leftarrow \mathtt{ref}(0)\, \mathtt{in}\, \lambda f.x := 1; f(); !x$ and $e_2 = \lambda f.f(); 1$.

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 $\mathtt{unit} \xrightarrow{\varepsilon} \mathtt{unit}$ with effects ε, then e_1 has type $(\mathtt{unit} \xrightarrow{\varepsilon} \mathtt{unit}) \xrightarrow{\varepsilon, rd_\mathsf{r}, wr_\mathsf{r}} \mathtt{int}\ \&\ al_\mathsf{r}$, while e_2 has type $(\mathtt{unit} \xrightarrow{\varepsilon} \mathtt{unit}) \to \mathtt{int}\ \&\ \varepsilon$. Notice that ε may contain rd_r or wr_r or both. Moreover, assume that an abstract location in region r looks at a single concrete location l:

$\mathfrak{l}^R = \{(\mathsf{h}, \mathsf{h}') \mid \mathsf{h}(\mathsf{l}) = \mathsf{h}'(\mathsf{l})\}$

$\mathfrak{l}^G = \{(\mathsf{h}, \mathsf{h}_1) \mid \mathsf{h}_1(\mathsf{l}) = 1 \wedge \forall \mathsf{l}' \neq \mathsf{l}.\mathsf{h}(\mathsf{l}') = \mathsf{h}_1(\mathsf{l}')\} \cup \{(\mathsf{h}, \mathsf{h}) \mid \mathsf{h} : \mathfrak{l}^R\}$

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 $[\![e_1]\!]\mathsf{w}\mathsf{h} = (\mathsf{w} \uplus \mathsf{w}_1 \uplus \mathsf{w}_r, u_1, \mathsf{h}_1, \mathsf{a}_1)$ and $[\![e_2]\!]\mathsf{w}\mathsf{h} = (\mathsf{w} \uplus \mathsf{w}_1, u_1, \mathsf{h}_2, \mathsf{a}_2)$. We construct a pullback square $\mathsf{w} \uplus \mathsf{w}_1 \uplus \mathsf{w}_r \diamond \mathsf{w} \uplus \mathsf{w}_1$ such that the values a_1 and a_2 are equal in its apex and h_1 and h_2 are equal in its low point. The value $\mathsf{a}_1 = 1$ because the value in l written to 1 and the function f when it is called cannot modify the value of l because $wr_\mathsf{r} \in \varepsilon$ and the guarantee relation defined above specifies that once the value of l is set to 1 it cannot be modified. We also have $\mathsf{a}_2 = 1$ trivially. Hence a_1 and a_2 are equal in the apex of the pullback square $\mathsf{w} \uplus \mathsf{w}_1 \uplus \mathsf{w}_r \diamond \mathsf{w} \uplus \mathsf{w}_1$. Similarly, h_1 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:

$e_1 = \mathtt{let}\, x \Leftarrow \mathtt{ref}(0)\, \mathtt{in}\, \lambda f.x := 0; f(); x := 1; f(); !x$

$e_2 = \lambda f.f(); f(); 1.$

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 follows in a similar way.

Callback with Lock Example We prove the equivalence of the following programs [1, 19]:

$e_1 = \texttt{let}\ b \Leftarrow \texttt{ref}(\texttt{true})\ \texttt{in}\ \texttt{let}\ x \Leftarrow \texttt{ref}(0)\ \texttt{in}$
$\langle \lambda f.\texttt{if}\ !b\ \texttt{then}\ (b := \texttt{false};$
$f(); x :=\! !x + 1; b := \texttt{true})\ \texttt{else}\ (), \lambda\_.!x\rangle$
$e_2 = \texttt{let}\ b \Leftarrow \texttt{ref}(\texttt{true})\ \texttt{in}\ \texttt{let}\ x \Leftarrow \texttt{ref}(0)\ \texttt{in}$
$\langle \lambda f.\texttt{if}\ !b\ \texttt{then}\ (b := \texttt{false};$
$\texttt{let}\ n \Leftarrow !x\ \texttt{in}\ f(); x := n + 1; b := \texttt{true})\ \texttt{else}\ (), \lambda\_.!x\rangle$.

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 `false` and only after the call-back is called and the value in x is incremented is b set again to `true`. However, the implementation of the increment function is different. While e_1 invokes the call-back $f()$ 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 $((\texttt{unit} \xrightarrow{\varepsilon} \texttt{unit}) \xrightarrow{\varepsilon, wr_\mathsf{r}, rd_\mathsf{r}} \texttt{unit}) \times (\texttt{unit} \xrightarrow{rd_\mathsf{r}} \texttt{unit})\ \&\ al_\mathsf{r}$, where ε may contain the effects $wr_\mathsf{r}, rd_\mathsf{r}$. In particular, the location l looks at two concrete locations l_b and l_x (storing x and b):

$\mathsf{I}^R = \{(\mathsf{h}, \mathsf{h}') \mid \mathsf{h}(\mathsf{l}_b) = \mathsf{h}'(\mathsf{l}_b) \wedge \mathsf{h}(\mathsf{l}_x) = \mathsf{h}'(\mathsf{l}_x)\}$

$\mathsf{I}^G = \{(\mathsf{h}, \mathsf{h}_1) \mid \mathsf{h}(\mathsf{l}_b) = \texttt{false} \Rightarrow \mathsf{h}_1 = \mathsf{h}$ and
$(\mathsf{h}(\mathsf{l}_b) = \texttt{true} \wedge \mathsf{h}_1(\mathsf{l}_x) = i \wedge \mathsf{h}(\mathsf{l}_x) = j) \Rightarrow$
$[(\mathsf{h}_1(\mathsf{l}_b) = \texttt{true}) \wedge (j \leq i) \wedge (\forall \mathsf{l}' \notin \{\mathsf{l}_x, \mathsf{l}_b\}.\mathsf{h}(\mathsf{l}') = \mathsf{h}_1(\mathsf{l}'))]\}$

First, notice that indeed the two programs above have the effect wr_r. The increment of x is allowed by I^G, as b is initially `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 effect in the region r, *i.e.*, $wr_\mathsf{r} \in \varepsilon$, it cannot change the value of x. This is because when the function is called, the value stored in b is `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.g.*, that our overlapping references v_{or} 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 $|A\mathsf{w}|$ 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 Π-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

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

[2] G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *J. Funct. Program.*, 13(2):261–293, 2003.

[3] N. Benton, L. Beringer, M. Hofmann, and A. Kennedy. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*, 2009.

[4] N. Benton, M. Hofmann, and V. Nigam. Proof-relevant logical relations for name generation. In *TLCA*, 2013.

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

[6] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*, 2007.

[7] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In *APLAS*, volume 4279 of *LNCS*, 2006.

[8] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *ICFP*, 1998.

[9] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, volume 3461 of *LNCS*, 2005.

[10] L. Birkedal, A. Carboni, G. Rosolini, and D. S. Scott. Type theory via exact categories. In *LICS*, 1998.

[11] L. Birkedal, K. Stovring, and J. Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theor. Comp. Sci.*, 411:4102–4122, 2010.

[12] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208(6), 2010.

[13] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.

[14] A. Carboni, P. J. Freyd, and A. Scedrov. A categorical approach to realizability and polymorphic types. In *MFPS, LNCS 298*, 1987.

[15] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. H. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw., Pract. Exper.*, 35(6):583–599, 2005.

[16] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*. Cambridge University Press, 1998.

[17] T. Dinsdale-Young, P. Gardner, and M. J. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, volume 6217 of *LNCS*, 2010.

[18] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

[19] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proc. ICFP, ACM*, pages 143–156, 2010.

[20] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, pages 233–246, 2005.

[21] M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[22] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, 1986.

[23] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.

[24] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and kripke logical relations. In J. Field and M. Hicks, editors, *POPL*, pages 59–72. ACM, 2012.

[25] J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, volume 7211 of *LNCS*, 2012.

[26] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL*, 2012.

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

[28] N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.

[29] N. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP*, 2012.

[30] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[31] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer, 2nd edition, Sept. 1998.

[32] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[33] F. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, CMU, 1982.

[34] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*, pages 227–273. Cambridge University Press, 1998.

[35] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[36] A. M. Pitts and I. D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or what's new? In *MFCS*, volume 711 of *LNCS*, 1993.

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

[38] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3), 2011.

[39] I. Stark. *Names and Higher-Order Functions*. PhD thesis, U. Cambridge, 1994.

[40] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *POPL*, 2005.

[41] J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In *ICFP*, 2011.

[42] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *LNCS*, 2007.

[43] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.