Abstract Effects and Proof-Relevant Logical Relations

Nick Benton, Martin Hofmann, and Vivek Nigam

Microsoft Research, LMU Munich and Federal University of Paraíba

Abstract. We introduce 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 well-known problems caused by the use of existential quantification over future worlds in traditional Kripke logical relations: failure of admissibility, and spurious functional dependencies.
We illustrate the novel format with two applications: a direct-style validation of Pitts and Stark's equivalences for "new" and 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 soundly to validate more effect-based program equivalences than would be possible with traditional effect systems.

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-, Java-, or assembly-like, but the common source of trickiness is the way effectful operations may be *partially* encapsulated behind higher-order abstractions. Problems in semantics and verification of effectful languages are often addressed using a range of common techniques that includes separation and Kripke logical relations (KLRs). The particular problem motivating the development of the proof-relevant form of KLR introduced here is that of giving a semantics to effect systems that accounts for partial encapsulation, though the general construction is more broadly applicable. As we will see, direct semantic reasoning in our model (as opposed to generic reasoning based on refined types) also allows many of the trickiest known equivalences concerning encapsulated store to be proved.

Effect systems [16] refine conventional types by tracking upper bounds on the side-effects of expressions. A series of papers, by ourselves and others [19, 5, 4, 6, 30], have explored the semantics of effect systems for mutable state, addressing not merely the correctness of analyses, but also the soundness of effect-dependent optimizations and refactorings. 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. Our primary interest is not syntactic rules for type assignment, but rather semantic interpretations

of effect-refined types that can justify such equivalences. Types provide a common interface language that can be used 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.

A key notion in compositional reasoning about 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 naively understood in terms of sets of locations. A memory allocator, for example, guarantees that its own private datastructures, memory belonging to clients, and any freshly-allocated block inhabit mutually disjoint sets of locations. Since the introduction of fractional permissions, separation logics often go 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' [20], 'fictional' [13, 18], 'subjective' [22], or 'superficial' [21] separation aims to let custom notions of separable resource be used and combined modularly. This paper presents a semantics for effect systems supporting fictional, or 'abstract', notions of both effects and separation.

We previously interpreted effect-refined types for stateful computations as binary relations, defined via preservation of particular sets of store relations. This already provides some abstraction. For example, a function that reads a reference, but whose result is independent of the value read can soundly be counted as pure (contrasting with models that instrument the concrete semantics). Our models also validated the masking rule, allowing certain non-observable effects not to appear in annotations. But here we go further, generalizing the interpretation of regions to partial equivalence relations (PERs). This allows, for example, a lookup function for a set ADT to be assigned a read-but-not-write effect, even if the concrete implementation involves non-observable writes to rebalance an internal datastructure. Roughly, there is a PER that relates two heaps iff they contain well-formed datastructures representing the same mathematical set, and the ADT operations respect this PER: looking up equal values in related heaps yields equal booleans, adding equal values in related heaps yields new related heaps, and so on. A mutating operation need only be annotated with a write effect if the updated heap is potentially in a different equivalence class from the original one. In fact, we further improve previous treatments of write effects, via a 'guarantee' condition that explicitly captures allowable local updates. Surprisingly, this allows the update and remove operations for our set ADT to be flagged with *just* a write effect, despite the fact that the final state of the set depends on the initial one, exploiting the idempotence of the updates and validating many more useful program transformations.

Moving to PERs also allows us to revisit the notion of separation, permitting distinct abstract locations, or regions, to refer to PERs whose footprints overlap, albeit non-observably, in memory. A module may, for example, implement two distinct logical references using a single physical location containing a coding (e.g. 2^i3^j) of a pair (i, j) of integers. Or a resource allocator can keep logically separated tokens tracking each allocated resource, acting as permissions for deallocation, in a shared datastructure such as a bitmap or linked list (a well-known problem in modular separation [21]). The innovation here is a notion of independence of PERs, capturing the situation where intersection of PERs yields a cartesian product of quotients of the heap.

$$
\begin{array}{l}
(T_\varepsilon Q)_{\mathsf{w}} = QPER(\{(f, f') \mid \mathsf{h}, \mathsf{h}' \models \mathsf{w} \Rightarrow \\
\quad \forall R \in \mathcal{R}_\varepsilon(\mathsf{w}).\mathsf{h} R \mathsf{h}' \Rightarrow \mathsf{h}_1 R \mathsf{h}'_1 \wedge \\
\quad \exists \mathsf{w}_1.(\mathsf{w}_1(r) \neq \emptyset \Rightarrow r \in \mathrm{als}(\varepsilon)) \wedge \mathsf{h}_1, \mathsf{h}'_1 \models \mathsf{w} \otimes \mathsf{w}_1 \wedge \\
\quad \mathsf{h}_1 \sim_{\mathsf{w}_1} \mathsf{h}'_1 \wedge (v, v') \in Q_{\mathsf{w} \otimes \mathsf{w}_1} \\
\quad \text{where } (\mathsf{h}_1, v) = f\mathsf{h} \text{ and } (\mathsf{h}'_1, v') = f'\mathsf{h}'\})
\end{array}
$$

Fig. 1: Earlier Kripke logical relation, extract

The ideas sketched above are intuitively rather compelling, but formally integrating them into the form of KLR we had previously used for effect systems turns out to be remarkably hard. Figure 1 shows a (tweaked) extract from an earlier paper [4]. Here a world w is just a finite partial bijection between locations, with region-coloured links; $\mathsf{h}, \mathsf{h}' \models \mathsf{w}$ simply means that for each link $(\mathsf{l}, \mathsf{l}') \in \mathsf{w}$, $\mathsf{l} \in \mathrm{dom}(\mathsf{h})$ and $\mathsf{l}' \in \mathrm{dom}(\mathsf{h}')$. Two computations $f, f' : \mathbb{H} \rightharpoonup \mathbb{H} \times \mathbb{V}$, where $\mathbb{H}, \mathbb{V}$ are sets of heaps and values, respectively, are in the relation $(T_\varepsilon Q)_{\mathsf{w}}$, where ε is an effect and the relation Q interprets a result type, if they preserve all heap relations R in a set depending on ε and w, and there exists *some* disjoint world extension w_1 such that the new heaps are equal on the domain of w_1, and the result values are Q-related at the extended world $\mathsf{w} \otimes \mathsf{w}_1$.

The problematic part is the existential quantification over world extensions – the $\exists \mathsf{w}_1$ on the third line – allowing for the computations to allocate fresh locations. This pattern of quantification occurs in many accounts of generativity, but the dependence of w_1 on both h and h' creates serious problems if 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 f', is started; the existential then produces a *different* extension w_2 that is not at all related, even on the side of f where the heap stays the same, to the w_1 with which one started. The case of bijections, where h_1 depends only on h (not on h'), allows one to deduce sufficient information about the domain of w_1 from the clause $\mathsf{h}_1, \mathsf{h}'_1 \models \mathsf{w} \otimes \mathsf{w}_1$, but this breaks down in the more abstract setting.

To fix this problem, we here take the rather novel step of replacing 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 an existential with a Σ-type. A statement like $(f, f') \in T_\varepsilon[\![A]\!]$ is no longer just a proposition, but we rather have a "set of proofs" $T_\varepsilon[\![A]\!](f, f')$ which in particular contains the aforementioned Skolem functions. We use an explicit version of the exact-completion [10, 8] akin to and motivated by "setoid" or groupoid interpretations of type theory [17, 3, 33] to make these ideas both rigorous and more general.

Passing from relations to proof-relevant setoids also solves other problems. Existential quantification fails to preserve admissibility of relations, needed to deal with general recursion, and also fails to preserve 'PERness'. The '$QPER(\cdot)$' operation in Figure 1 explicitly applies an admissible and (variant) PER closure operation; this works technically, but is very awkward to use. We do not need such a closure here. Step indexing [2, 30] and the use of continuations [27] can also deal with admissibility. However, step-indexing is inherently operational, whilst continuations lose sufficient abstraction to break some program equivalences, including commuting computations. Our third way, using setoids, is pleasantly direct. Finally, allocation effects are handled differently from reading and writing by the relation in Figure 1, being wired into the quantification rather than treated more abstractly by relation preservation. Our setoid-based formulation uses uniform machinery to treat all effects.

We start by reviewing some preliminary definitions on syntax and semantics of programs in Section 2. Section 3 introduces setoids, which is the setting in which we

specify in Section 4 the typed semantics and introduce the notion of abstract effects. In Section 5 we describe proof-relevant logical relations, prove the fundamental theorem and define observational equivalence. Section 6 demonstrates a number of program equivalences that can be shown by using proof-revelant logical relations. We conclude and discuss future work in Section 7.

Note: We have elided many proofs, details of constructions and examples. This longer version of the paper includes some of this material in an appendix.

2 Syntax and Semantics

We will interpret effect-refined types over a somewhat generic, untyped denotational model for stateful computations in the category of predomains (ω-cpos). We also introduce a meta-language [24], providing concrete syntax for functions in the model. We omit the standard details of interpreting CBV programming languages via such a metalanguage, or proofs of adequacy, relating the operationally induced observational (in)equivalence to (in)equality in the model.

Denotational model We assume predomains $\mathbb{V}$ and $\mathbb{H}$ modelling values and heaps, respectively. As much of the metatheory does not rely on the finer details of how these predomains are defined, we axiomatise the properties we use. Firstly, we assume the existence of a set of (concrete) locations $\mathbb{L}$ and for each $\mathsf{h} \in \mathbb{H}$ a finite set $\mathrm{dom}(\mathsf{h}) \subseteq \mathbb{L}$. We also assume a constant $\emptyset \in \mathbb{H}$, the empty heap. If $\mathsf{h} \in \mathbb{H}, \mathsf{l} \in \mathrm{dom}(\mathsf{h})$, then $\mathsf{h}(\mathsf{l}) \in \mathbb{V}$. If $v \in \mathbb{V}, \mathsf{h} \in \mathbb{H}, \mathsf{l} \in \mathrm{dom}(\mathsf{h})$ then $\mathsf{h}[\mathsf{l} \mapsto v] \in \mathbb{H}$; finally $\mathit{new}(\mathsf{h}, v)$ yields a pair $(\mathsf{l}, \mathsf{h}')$ where $\mathsf{l} \in \mathbb{L}$ and $\mathsf{h}' \in \mathbb{H}$. These three operations are continuous, in particular, $\mathsf{h} \leq \mathsf{h}' \Rightarrow \mathrm{dom}(\mathsf{h}) \subseteq \mathrm{dom}(\mathsf{h}')$ and the following axioms 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 $\mathit{new}(\mathsf{h}, v) = (\mathsf{l}, \mathsf{h}')$ then $\mathrm{dom}(\mathsf{h}') = \mathrm{dom}(\mathsf{h}) \cup \{\mathsf{l}\}$ and $\mathsf{l} \notin \mathrm{dom}(\mathsf{h})$ and $\mathsf{h}'(\mathsf{l}) = v$. Given $\mathbb{V}$ this abstract datatype can be implemented in a number of ways, e.g., as finite maps. We define the domain of computations $\mathbb{C}$ to be partial continuous functions from $\mathbb{H}$ to $\mathbb{H} \times \mathbb{V}$, the bottom element being the everywhere undefined function.

We assume that $\mathbb{V}$ embeds tuples of values, i.e., if $v_1, \ldots, v_n \in \mathbb{V}$ then $(v_1, \ldots, v_n) \in \mathbb{V}$ and it is possible to tell whether a value is of that form and in this case to retrieve the components. We also assume that $\mathbb{V}$ embeds continuous functions $f : \mathbb{V} \to \mathbb{C}$, i.e., if f is such a function then $\mathit{fun}(f) \in \mathbb{V}$ and, finally, locations are also values, i.e. if $\mathsf{l} \in \mathbb{L}$ then $\mathit{loc}(\mathsf{l}) \in \mathbb{V}$ and one can tell whether a value is a location or a function. A canonical example of such a $\mathbb{V}$ is the least solution to the predomain equation with $\mathbb{C} = \mathbb{H} \rightharpoonup \mathbb{H} \times \mathbb{V}$ and $\mathbb{V} \simeq \mathit{int}(\mathbb{Z}) + \mathit{fun}(\mathbb{V} \to \mathbb{C}) + \mathit{loc}(\mathbb{L}) + \mathbb{V}^*$.

Syntax The syntax of untyped values and computations is:

$$v ::= x \mid () \mid c \mid (v_1, v_2) \mid v.1 \mid v.2 \mid \mathtt{rec}\ f\ x = t$$

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

Here, x ranges over variables and c over constant symbols, each of which has an associated interpretation $[\![c]\!] \in \mathbb{V}$; these include numerals $\underline{n}$ with $[\![\underline{n}]\!] = \mathit{int}(n)$, arithmetic operations and so on. $\mathtt{rec}\ f\ x = t$ 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 intialised 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.

Semantics The untyped semantics of values $[\![v]\!] \in \mathbb{V} \to \mathbb{V}$ and terms $[\![t]\!] \in \mathbb{V} \to \mathbb{C}$ are defined by an entirely standard mutual induction, using least fixed points to interpret recursive functions, projection from tuples for variables and so on.

Types Types are given by the grammar: $\tau ::= \mathtt{unit} \mid \mathtt{int} \mid A \mid \tau_1 \times \tau_2 \mid \tau_1 \xrightarrow{\varepsilon} \tau_2$, where A ranges over semantically defined basic types (see Def. 11). These contain reference types possibly annotated with regions and abstract 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 2. We do not bake in type rules for constants and effectful operations but, for a given semantic interpretation of types, we will be able to justify adding further rules for these primitives and, more importantly, for more complex expressions involving them. (The rules given here incorporate subeffecting; we expect our semantics to extend to more general subtyping.)

Equations Figure 3 outlines a core equational theory for the metalanguage. The full theory includes congruence rules for all constructs (like that given for `rec`), all the usual beta and eta laws and commuting conversions for conditionals as well as for `let`. We give a semantic interpretation of typed equality judgements 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. The core theory then allows one to deduce new semantic equalities from already proven ones. The equations are typed: a derivation $\mathcal{D}$ of $\Gamma \vdash t = t' : \tau \;\&\; \varepsilon$ is canonically associated with typing derivations $\mathcal{D}.1$ and $\mathcal{D}.2$ of $\Gamma \vdash t : \tau \;\&\; \varepsilon$ and $\Gamma \vdash t' : \tau \;\&\; \varepsilon$, respectively (but note we can semantically justify extending the type rules). The interpretation of $\mathcal{D}$ will be a proof object certifying that the interpretations of $\mathcal{D}.1$ and $\mathcal{D}.2$ are semantically equal which then implies (Theorem 3) typed observational equivalence of t and t'.

$$\frac{}{\Gamma \vdash \underline{n} : \mathtt{int}} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \tau \;\&\; \emptyset} \qquad \frac{\Gamma \vdash e : \tau \;\&\; \varepsilon_1 \quad \varepsilon_1 \subseteq \varepsilon_2}{\Gamma \vdash e : \tau \;\&\; \varepsilon_2} \qquad \frac{\Gamma \vdash v : \tau_1 \times \tau_2}{\Gamma \vdash v.i : \tau_i}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \xrightarrow{\varepsilon} \tau_2 \quad \Gamma \vdash v_2 : \tau_1}{\Gamma \vdash v_1\ v_2 : \tau_2 \;\&\; \varepsilon} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{\Gamma \vdash v : \mathtt{int} \quad \Gamma \vdash e_1 : \tau \;\&\; \varepsilon \quad \Gamma \vdash e_2 : \tau \;\&\; \varepsilon}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \;\&\; \varepsilon}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \;\&\; \varepsilon \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2 \;\&\; \varepsilon}{\Gamma \vdash \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2 : \tau_2 \;\&\; \varepsilon} \qquad \frac{\Gamma, f{:}\tau_1 \xrightarrow{\varepsilon} \tau_2, x{:}\tau_1 \vdash e : \tau_2 \;\&\; \varepsilon}{\Gamma \vdash \mathtt{rec}\ f\ x = e : \tau_1 \xrightarrow{\varepsilon} \tau_2}$$

Fig. 2: Core rules for effect typing

2.1 Some example programs

Dummy allocation Define *dummy* as $[\![\lambda f.\lambda x.\mathtt{let}\ d \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ f\ x]\!]$, so $dummy(f)$ behaves like f but makes an allocation whose result is discarded. We will be able to show that $dummy(f)$ displays no more abstract effects than f, so that whatever program transformation f can participate in, $dummy(f)$ can as well.

Memoisation Let *memo* be the memoizing functional

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

$$\frac{\Gamma \vdash t : \tau \;\&\; \varepsilon}{\Gamma \vdash t = t : \tau \;\&\; \varepsilon} \quad \frac{\Gamma \vdash t = t' : \tau \;\&\; \varepsilon}{\Gamma \vdash t' = t : \tau \;\&\; \varepsilon} \quad \frac{\Gamma \vdash t = t' : \tau \;\&\; \varepsilon \quad \Gamma \vdash t' = t'' : \tau \;\&\; \varepsilon}{\Gamma \vdash t = t'' : \tau \;\&\; \varepsilon} \quad \frac{\Gamma \vdash v = v' : \tau}{\Gamma \vdash v = v' : \tau \;\&\; \emptyset}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2).i = v_i : \tau_i} \quad \frac{\Gamma, f : \tau_1 \xrightarrow{\varepsilon} \tau_2, x{:}\tau_1 \vdash t = t' : \tau_2 \;\&\; \varepsilon}{\Gamma \vdash (\mathtt{rec}\ f\ x = t) = (\mathtt{rec}\ f\ x = t') : \tau_1 \xrightarrow{\varepsilon} \tau_2} \quad \frac{\Gamma \vdash v : \tau_1 \times \tau_2}{\Gamma \vdash v = (v.1, v.2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v : \tau_1 \;\&\; \varepsilon \quad \Gamma, x : \tau_1 \vdash t : \tau_2 \;\&\; \varepsilon}{\Gamma \vdash \mathtt{let}\ x \Leftarrow v\ \mathtt{in}\ t = t[v/x] : \tau_2 \;\&\; \varepsilon} \quad \frac{\Gamma, f : \tau_1 \xrightarrow{\varepsilon} \tau_2, x{:}\tau_1 \vdash t : \tau_2 \;\&\; \varepsilon \quad \Gamma \vdash v : \tau_1}{\Gamma \vdash (\mathtt{rec}\ f\ x = t)\, v = t[v/x, (\mathtt{rec}\ f\ x = t)/f] : \tau_2 \;\&\; \varepsilon}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \;\&\; \varepsilon \quad \Gamma \vdash t_2 : \tau_2 \;\&\; \varepsilon \quad \Gamma, x : \tau_2, y : \tau_1 \vdash t_3 : \tau_3 \;\&\; \varepsilon}{\Gamma \vdash \mathtt{let}\ x \Leftarrow (\mathtt{let}\ y \Leftarrow t_1\ \mathtt{in}\ t_2)\ \mathtt{in}\ t_3 = \mathtt{let}\ y \Leftarrow t_1\ \mathtt{in}\ \mathtt{let}\ x \Leftarrow t_2\ \mathtt{in}\ t_3 : \tau_3 \;\&\; \varepsilon}$$

Fig. 3: Basic equational theory (extract)

where $t_1; t_2 = \mathtt{let}\ \_ \Leftarrow t_1\ \mathtt{in}\ t_2$ is sequential composition and *eq* is an integer equality constant. We can justify 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. This was not justified by our previous model [4].

Set factory The next, more complicated, example is a program that can create and manipulate sets implemented as linked lists.

If $\mathsf{l} \in \mathbb{L}$ and $\mathsf{h} \in \mathbb{H}$ and U is a finite set of integers and P is a finite subset of $\mathbb{L}$ define $S(\mathsf{l}, \mathsf{h}, U, P)$ to mean that in h location l points to a linked list of integer values occupying at most the locations in P (the "footprint") and so that the set of these integer values is U. So, for example, if $\mathsf{h}(\mathsf{l}) = loc(\mathsf{l}_1)$ and $\mathsf{h}(\mathsf{l}_1) = (int(1), loc(\mathsf{l}_2))$ and $\mathsf{h}(\mathsf{l}_2) = (int(1), int(0))$ then $S(\mathsf{l}, \mathsf{h}, \{1\}, \{\mathsf{l}_1, \mathsf{l}_2\})$ holds.

For each location l define functions mem_{l}, add_{l}, rem_{l} so that $mem_{\mathsf{l}}(int(i))$ checks whether i occurs in the list pointed to by l, returning $int(1)$ iff yes, and—for the fun of it—removes all duplicates in that list and relocates some of its nodes. Thus, in particular, if $mem_{\mathsf{l}}(int(i))(\mathsf{h}) = (\mathsf{h}_1, v)$ then if $S(\mathsf{l}, \mathsf{h}, U, P)$ one has $S(\mathsf{l}, \mathsf{h}_1, U, P')$ for some P' where $P' \subseteq P \cup (\mathrm{dom}(\mathsf{h}_1) \setminus \mathrm{dom}(\mathsf{h}))$ and $v = int(1)$ iff $i \in U$.

The function add_{l} adds its integer argument to the set, and rem_{l} removes it, each possibly making "optimizations" similar to mem_{l}.

Now consider a function *setfactory* returning upon each call a fresh location l and a the tuple of functions $(mem_{\mathsf{l}}, add_{\mathsf{l}}, rem_{\mathsf{l}})$. We will be able to justify the following semantic typing for *setfactory*:

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

which expresses that *setfactory*() allocates in some (possibly fresh) region r and returns operations that only read r (the first one) or write in r (the second and third one) even though, physically, all three functions read, write, and allocate.

Thus, these functions can participate in corresponding effect-dependent program equivalences, in particular, two successive *mem* operations may be swapped and duplicated; identical updates may even be contracted.

Interleaved Dummy allocation Consider the following example, which looks similar to the Dummy example above, but where the dummy allocation happens after a proper allocation:

$$e_1 = \mathtt{let}\ p \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \mathtt{let}\ d \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ e; !p \quad \text{and} \quad e_2 = \mathtt{let}\ p \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ e; !p.$$

Here d is not free in e, but p may be free. This simple difference leads to many problems when attempting to prove their equivalence. We sketch them below to also motivate our technical solution introduced formally in the following Sections.

As normally done the evolution of the heaps can be formally captured by using Kripke models, where, intuitively, a world contains the set of locations allocated by programs. Whenever there is an allocation, we advance from the current world w to a world w_1, which contains some fresh locations. However, we do not have control over this evolution. In our example, assume that the programs above start at the same world w. The allocation of the proper location, p, in e_1 and in e_2 will yield two different extensions $\mathsf{w} \to \mathsf{w}_1$ and $\mathsf{w} \to \mathsf{w}'_1$, where some concrete locations, l_1 and l_2, are allocated respectively. In fact, w_1 and w'_1 may even contain other locations that are not used by the computations. For proving the equivalence between these programs, we need a way to capture that l_1 and l_2 are equivalent, without requiring to identify the other locations not used by computations.

Our solution is to use *pullback squares* as proofs. Their shape is depicted in Figure 4. where $\underline{\mathsf{w}}$ and $\overline{\mathsf{w}}$ are called, respectively, the *low point* and *apex* of the square. It helps to interpret $\overline{\mathsf{w}}$ as a superset of $\mathsf{w}_1 \cup \mathsf{w}'_1$, that is, a world containing all the locations mentioned in w_1 and w'_1, even the locations not used by computations, while $\underline{\mathsf{w}} = \mathsf{w}_1 \cap \mathsf{w}_2$ (modulo renaming of location names) is a world containing only the locations that need to be identified. Intuitively, the low point is the part of the proof showing that resulting heaps of computations are equivalent. This is formalized by Definition 13. In the example above, the low point is a world where l_1 and l_2 are shown to be equivalent. The remaining locations in w_1 and w'_1 that are not used by computations may be ignored, that is, not be contained in $\underline{\mathsf{w}}$. The apex, $\overline{\mathsf{w}}$, on the other hand, is the part of the proof showing that the corresponding *values* resulting from computations, $!p$ in the example above, are indeed equivalent (see again Definition 13).

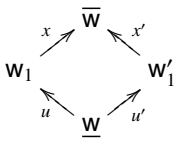

Fig. 4: Pullback square.

3 Setoids

We define the *category of setoids* as the exact completion of the category of predomains, see [10, 8]. We give here an elementary description using the language of dependent types. 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 $t_A : \Pi x, y, z.A(x, y) \times A(y, z) \to A(x, z)$. If $p \in A(x, y)$ we may write $p : x \sim y$ or simply $x \sim y$. We also omit $|-|$ wherever appropriate. We remark that "setoids" also appear in constructive mathematics and formal proof, see e.g., [3], but the proof-relevant nature of equality proofs is not exploited there and everything is based on sets (types) rather than predomains. 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 $\mu : \Pi a \in |A|.B(f(a), g(a))$.

Proposition 1. *The category of setoids is cartesian closed; moreover, if D is a setoid such that $|D|$ has a least element $\perp$ and there is also a least proof $\perp \in D(\perp, \perp)$ then there is a morphism of setoids $Y : [D \to D] \to D$ satisfying the usual fixpoint equations.*

3.1 Pullback squares A morphism u in a category $\mathbf{W}$ 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 $v' = u't$. We 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 such a pullback square. 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 written $\underline{\mathsf{w}}$. A pullback square $xu = x'u'$ is *minimal* if whenever $fx = gx$ and $fx' = gx'$ then $f = g$, in other words, x and x' are *jointly epic*. 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.

Definition 1 (Category of worlds). *A category* $\mathbf{W}$ *is a* category of worlds *if it has pullbacks and every span can be completed to a minimal pullback square and all morphisms are monomorphisms.*

Example 3.1 The category of sets and injections is a category of worlds. Given $f : X \to Z$ and $g : Y \to Z$, we form their pullback as $X \xleftarrow{f^{-1}} fX \cap gY \xrightarrow{g^{-1}} Y$. This is minimal when $fX \cup gY = Z$. Conversely, given a span $Y \xleftarrow{f} X \xrightarrow{g} Z$, we can complete to a minimal pullback by

$$(Y \setminus fX) \uplus fX \xrightarrow{[in_1, in_3 \circ f^{-1}]} (Y \setminus fX) + (Z \setminus gX) + X \xleftarrow{[in_2, in_3 \circ g^{-1}]} (Z \setminus gX) \uplus gX$$

where $[-,-]$ is case analysis on the disjoint union $Y = (Y \setminus fX) \uplus fX$.

Given an arbitrary category $\mathbf{C}$, the category of worlds $\mathbf{W_C}$ has objects pairs (X, f) where X is a set and $f : X \to |\mathbf{C}|$ is an X-indexed family of $\mathbf{C}$-objects. A morphism from (X, f) to (Y, g) is an injective function $u : X \to Y$ and a family of isomorphisms $\varphi_x : f(x) \simeq g(u(x))$. The first components of the pullbacks and minimal pullbacks are constructed as in the previous example. □

We write $r(\mathsf{w})$ for $\mathsf{w}{}^{1}_{1}\Diamond^{1}_{1}\mathsf{w}$ and $s({}^{x}_{u}\Diamond^{x'}_{u'}) = {}^{x'}_{u'}\Diamond^{x}_{u}$ and $t({}^{x}_{u}\Diamond^{x'}_{u'}, {}^{y}_{v}\Diamond^{y'}_{v'}) = {}^{zx}_{z'y'}\Diamond^{ut}_{v't'}$ where z, z', t, t' are chosen so that all four participating squares are pullbacks.

3.2 Setoid-valued functors A functor A from a category of worlds $\mathbf{W}$ 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 $a \in A\mathsf{w}$ we may write $u.a$ or even ua for $Au(a)$ and likewise for proofs in $A\mathsf{w}$. Note that $(uv).a = u.(v.a)$.

Definition 2. *We call a functor* 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* $A\mathsf{w}{}^{Ax}_{Au}\Diamond^{Ax'}_{Au'}A\mathsf{w}'$ *is a pullback in Std. This means that there is a continuous function of type*

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

Thus, if two values $a \in A\mathsf{w}$ and $a' \in A\mathsf{w}'$ 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, a' arise. Intuitively, p.p.f.s will become the denotations of value types.

3.3 Fibred setoids In order to provide meanings for computation types we need a weaker variant of p.p.f., namely, *fibred setoids*. These lack the facility of transporting values along world morphisms but instead allow the proof-relevant comparison of values at different worlds provided the latter are related by a pullback square.

Definition 3. *A* fibred setoid *over a category of worlds* $\mathbf{W}$ *is given by a predomain* $T\mathsf{w}$ *for every* $\mathsf{w} \in \mathbf{W}$ *and for every pullback square* $\mathsf{w}\diamond\mathsf{w}'$ *and elements* $a \in T\mathsf{w}$ *and* $a' \in T\mathsf{w}'$ *a set* $T\diamond(a,a')$ *so that the set of tuples* (a,a',q) *with* $q \in T\diamond(a,a')$ *is a predomain with continuous projections.*

Next, we need continuous operations r,s,t *so that* $r(a) \in Tr(\mathsf{w})(a,a)$ *when* $a \in T\mathsf{w}$ *and* $s(q) \in Ts(\diamond)(a',a)$ *when* $q \in T\diamond(a,a')$ *and* $t(q,q') \in Tt(\diamond,\diamond')(a,a'')$ *when* $q \in T\diamond(a,a')$ *and* $q' \in T\diamond'(a',a'')$.

In addition, for any two isomorphic pullback squares $\diamond$ *and* $\diamond'$ *between* w *and* w' *there is a continuous operation of type* $\Pi a \in T\mathsf{w}.\Pi a' \in T\mathsf{w}'.T\binom{\diamond}{a}\binom{\diamond'}{a'} \to T'\binom{\diamond}{}\binom{a}{a'}$.

Finally, for each pullback square $\diamond = \mathsf{w}^x_u\diamond^{x'}_{u'}\mathsf{w}'$ *with apex* $\overline{\mathsf{w}}$ *and low point* $\underline{\mathsf{w}}$ *there is a continuous function of type*

$\Pi t \in T\mathsf{w}.\Pi t' \in T\mathsf{w}'.T\diamond(t,t') \to \Sigma \underline{t} \in T\underline{\mathsf{w}}.T^u_1\diamond^1_u(\underline{t},t) \times T^{u'}_1\diamond^1_{u'}(\underline{t},t')$

Note the similarity of the last operation to pullback-preservation.

Example 3.2 If A is a p.p.f., we obtain a fibred setoid $S(A)$ as follows: $S(A)\mathsf{w} = A\mathsf{w}$ and if $\mathsf{w}^x_u\diamond^{x'}_{u'}\mathsf{w}'$ with apex $\overline{\mathsf{w}}$, define the proof set $S(A)^x_u\diamond^{x'}_{u'}(a,a') = A\overline{\mathsf{w}}(x.a, x'.a')$. □

Definition 4. *A* morphism f *from fibred setoid* T *to fibred setoid* T' *is an equivalence class of pairs of continuous functions* $f_0 : \Pi\mathsf{w}.T\mathsf{w} \to T'\mathsf{w}$ *and* $f_1 : \Pi\mathsf{w},\mathsf{w}'.\Pi\mathsf{w}\diamond\mathsf{w}'.\Pi a \in T\mathsf{w}.\Pi a' \in T\mathsf{w}'.T\diamond(a,a') \to T'\diamond(f_0(\mathsf{w},a), f_0(\mathsf{w}',a'))$.

Two such pairs f, f' *are identified if there exists a continuous function that assigns to each* w *and* $a \in T\mathsf{w}$ *a proof* $\mu(a) \in Tr(\mathsf{w})(f_0(\mathsf{w},a), f'_0(\mathsf{w},a))$.

3.4 Contravariant functors and relations The role of the next concept is to give meaning to abstract stores.

Definition 5. *A contravariant functor* $\mathfrak{S}$ *from a category of worlds* $\mathbf{W}$ *to the category of setoids comprises for each* $\mathsf{w} \in \mathbf{W}$ *a* nonempty *setoid* $\mathfrak{S}\mathsf{w}$ *and for each morphism* $u : \mathsf{w}_0 \to \mathsf{w}$ *a setoid morphism* $\mathfrak{S}u : \mathfrak{S}\mathsf{w} \to \mathfrak{S}\mathsf{w}_0$ *such that* $u \mapsto \mathfrak{S}u$ *preserves identities and composition.*

If $\sigma \in \mathfrak{S}\mathsf{w}$ and $u : \mathsf{w}_0 \to \mathsf{w}$ we write $\sigma.u$ or σu for $\mathfrak{S}u(\sigma)$. Note that $\sigma.(uv) = (\sigma.u).v$. Intuitively, $\sigma.u$ can be interpreted as the abstract heap obtained by forgetting locations in σ that have been "allocated" by the world evolution specified by u, namely, those appearing in w and not in w_0.

Definition 6. *A contravariant functor* $\mathfrak{S}$ *preserves* minimal pullbacks *if whenever* $\mathsf{w}^x_{\underline{u}}\diamond^{x'}_{u'}\mathsf{w}'$ *with apex* $\overline{\mathsf{w}}$ *and low point* $\underline{\mathsf{w}}$ *is a* minimal pullback square *then the diagram* $\mathfrak{S}\mathsf{w} \;{}^{\mathfrak{S}u}_{\mathfrak{S}u'}\diamond^{\mathfrak{S}x}_{\mathfrak{S}x'}\; \mathfrak{S}\mathsf{w}'$ *is a pullback in Std.*

This means in particular that if $\sigma \in \mathfrak{S}\mathsf{w}, \sigma' \in \mathfrak{S}\mathsf{w}'$ and $\sigma.u \sim \sigma'.u'$ then there exists a "pasting" $\overline{\sigma} \in \mathfrak{S}\overline{\mathsf{w}}$ such that $\overline{\sigma}.x \sim \sigma$ and $\overline{\sigma}.x' \sim \sigma'$ and $\overline{\sigma}$ is unique up to $\sim$. Moreover the passage from the given data to $\overline{\sigma}$ and the witnessing proofs is continuous.

Definition 7. *A* relation R *on such a contravariant functor* $\mathfrak{S}$ *consists of an admissible subset* $R\mathsf{w} \subseteq \mathfrak{S}\mathsf{w} \times \mathfrak{S}\mathsf{w}$ *such that* $(\sigma,\sigma') \in R\mathsf{w}$ *and* $u : \mathsf{w}_0 \to \mathsf{w}$ *implies* $(\sigma.u, \sigma'.u) \in R\mathsf{w}_0$ *and if* $p : \sigma \sim \sigma_1$ *and* $p' : \sigma' \sim \sigma'_1$ *then* $(\sigma_1,\sigma'_1) \in R\mathsf{w}$*, as well.*

It would be natural to let relations be proof-relevant as well, but we refrain from doing so at this stage for the sake of simplicity.

4 Computational model

We use a setoid interpretation in order to justify nontrivial type-dependent observational equivalences for the language above. This interpretation is parametric over an *instantiation*, defined below.

Definition 8. *An* instantiation *comprises the following data.*
• *a category of worlds* $\mathbf{W}$*;*
• *a full-on-objects subcategory* $\mathbf{I}$ *of* inclusions *(in other words, a subset of the morphisms closed under composition and comprising the identities) with the property that every morphism u can be factored as $u = fi$ and $u = jg$ with f, g isomorphisms and i, j inclusions;*
• *a contravariant, minimal-pullback-preserving, functor* $\mathfrak{S}$ *from* $\mathbf{W}$ *to the category of setoids;*
• *for each* $\mathsf{w} \in \mathbf{W}$ *a relation* $\Vdash_{\mathsf{w}} \subseteq \mathbb{H} \times \mathfrak{S}\mathsf{w}$ *subject to the axiom that* $\mathsf{h} \Vdash_{\mathsf{w}} \sigma$ *and* $u \in \mathbf{I}(\mathsf{w}_0, \mathsf{w})$ *implies* $\mathsf{h} \Vdash_{\mathsf{w}_0} \sigma.u$*;*
• *a set of elementary effects* $\mathcal{E}$ *and for each effect* ε *a set* $\mathcal{R}(\varepsilon)$ *of relations on* $\mathfrak{S}$*. As usual, one defines effects as sets of elementary effects and extends* $\mathcal{R}$ *to all effects by* $\mathcal{R}(\emptyset)$ = *"all relations on* $\mathfrak{S}$ *(in the sense described in Section 3.4)" and* $\mathcal{R}(\varepsilon) = \bigcap_{\varepsilon_0 \in \varepsilon} \mathcal{R}(\varepsilon_0)$*.*

We give two examples of instantiations. The appendix contains a third example, mirroring our previous model [5].

4.1 Sets of locations In the first one, called *sets of locations*, worlds are finite sets of (allocated) locations (taken from $\mathbb{L}$) and their morphisms are injective functions with inclusions being actual inclusions. Abstract stores are given by $\mathfrak{S}\mathsf{w} = \{\mathsf{h} \mid \mathrm{dom}(\mathsf{h}) \supseteq \mathsf{w}\}$ with $\mathfrak{S}\mathsf{w}(\mathsf{h}, \mathsf{h}') = \star$, always, and $\mathfrak{S}u$ given by renaming locations.

We put $\mathsf{h} \Vdash_{\mathsf{w}} \mathsf{h}'$ whenever $\mathsf{h} = \mathsf{h}'$. We only have one elementary effect here, al, representing the allocation of one or more fresh names. Note that if R is a relation on $\mathfrak{S}$ then $R\mathsf{w}$ is either total or empty and if $u : \mathsf{w} \to \mathsf{w}'$ then $R\mathsf{w}' \neq \emptyset \Rightarrow R\mathsf{w} \neq \emptyset$. A relation R is in $\mathcal{R}(al)$ if for every inclusion $u : \mathsf{w} \to \mathsf{w}'$ one also has $R\mathsf{w} \neq \emptyset \Rightarrow R\mathsf{w}' \neq \emptyset$, thus R is oblivious to world extensions.

4.2 Abstract locations To formulate the second instantiation, called *Heap PERs*, we need the concept of an *abstract location* which generalises 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 bunch of (continuous) functions that model *writing to the* abstract location. These functions are closed under composition (thus form a category) and are idempotent in the sense of the PER modelling equality.

Thirdly, a "footprint" which is a heap-dependent set of physical locations which overapproximates the effect of "the guarantee" so as to enable the creation of fresh abstract locations not knowing the precise nature of the other abstract locations that are already there. (These footprints are very similar to accessibility maps, first introduced for reasoning in a model of state based on FM-domains [7].)

Definition 9. *An* abstract location $\mathfrak{l}$ *(on the chosen predomain* $\mathbb{H}$*) consists of the following data:*

- *a nonempty, admissible 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 set* $\mathfrak{l}^G$ *of continuous functions on* $\mathbb{H}$ *closed by composition, modelling the functions that "write only on* $\mathfrak{l}$*" leaving other locations alone (a "guarantee condition");*
- *a continuous function* $\mathfrak{l}^F : \Pi \mathsf{h} \in \mathbb{H}.\mathcal{P}(\mathrm{dom}(\mathsf{h}))$ *describing the "footprint" of the abstract location (where the ordering on the powerset* $\mathrm{dom}(\mathsf{h})$ *is of course discrete).*

subject to the conditions

- *if* $\iota \in \mathfrak{l}^G$ *and* $(\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R$ *then* $(\iota(\mathsf{h}), \iota(\mathsf{h}')), (\iota(\mathsf{h}), \iota(\iota(\mathsf{h}))), (\iota(\mathsf{h}'), \iota(\iota(\mathsf{h}'))) \in \mathfrak{l}^R$,
- *if* $\forall \mathsf{l} \in \mathfrak{l}^F(\mathsf{h}).\mathsf{h}_1(\mathsf{l}) = \mathsf{h}(\mathsf{l})$ *and* $\forall \mathsf{l} \in \mathfrak{l}^F(\mathsf{h}').\mathsf{h}'_1(\mathsf{l}) = \mathsf{h}'(\mathsf{l})$ *then* $(\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R$ *implies* $(\mathsf{h}_1, \mathsf{h}'_1) \in \mathfrak{l}^R$*; thus* $\mathfrak{l}^R$ *"looks" no further than the footprint;*
- *if* $\iota \in \mathfrak{l}^G$ *and* $\iota(\mathsf{h}) = \mathsf{h}_1$ *then* $\mathrm{dom}(\mathsf{h}) \subseteq \mathrm{dom}(\mathsf{h}_1)$ *and* $\mathsf{l} \in \mathrm{dom}(\mathsf{h}) \setminus \mathfrak{l}^F(\mathsf{h})$ *implies* $\mathsf{l} \notin \mathfrak{l}^F(\mathsf{h}_1)$ *and* $\mathsf{h}(\mathsf{l}) = \mathsf{h}_1(\mathsf{l})$.

Two abstract locations $\mathfrak{l}_1, \mathfrak{l}_2$ *are independent if*

- *for* $i = 1, 2$ *and* $\iota(\mathsf{h}) = \mathsf{h}_1$ *for* $\iota \in \mathfrak{l}_i^G$ *one has* $(\mathsf{h}, \mathsf{h}) \in \mathfrak{l}_i^R, (\mathsf{h}, \mathsf{h}') \in \mathfrak{l}_{3-i}^R \Rightarrow (\mathsf{h}_1, \mathsf{h}') \in \mathfrak{l}_{3-i}^R$ *and* $\mathsf{l} \in \mathrm{dom}(\mathsf{h}) \setminus \mathfrak{l}_{3-i}^F(\mathsf{h})$ *then* $\mathsf{l} \notin \mathfrak{l}_{3-i}^F(\mathsf{h}_1)$*;*
- *If* $(\mathsf{h}_1, \mathsf{h}_1) \in \mathfrak{l}_1^R$ *and* $(\mathsf{h}_2, \mathsf{h}_2) \in \mathfrak{l}_2^R$ *there exists* h *such that* $(\mathsf{h}, \mathsf{h}_1) \in \mathfrak{l}_1^R$ *and* $(\mathsf{h}, \mathsf{h}_2) \in \mathfrak{l}_2^R$*. (Amounting to* $\mathsf{h}/(\mathfrak{l}_1^R \cap \mathfrak{l}_2^R)$ *being a cartesian product of* $\mathsf{h}/\mathfrak{l}_1^R$ *and* $\mathsf{h}/\mathfrak{l}_2^R$*.)*

If $\mathfrak{l}_1, \mathfrak{l}_2$ *are independent, we form a joint location* $\mathfrak{l}_1 \otimes \mathfrak{l}_2$ *by* $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^R = \mathfrak{l}_1^R \cap \mathfrak{l}_2^R$ *and* $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^G = (\mathfrak{l}_1^G \cup \mathfrak{l}_2^G)^*$ *and* $(\mathfrak{l}_1 \otimes \mathfrak{l}_2)^F(\mathsf{h}) = \mathfrak{l}_1^F(\mathsf{h}) \cup \mathfrak{l}_2^F(\mathsf{h})$.

If $\mathsf{l} \in \mathbb{L}$ is a concrete location, we can define an abstract counterpart by putting $\mathsf{l}^R = \{(\mathsf{h}, \mathsf{h}') \mid \mathsf{h}(\mathsf{l}) = \mathsf{h}'(\mathsf{l})\}$ and l^G is the set with a write function for each value that may be stored in l. For instance, if l stores booleans, then l^G contains the functions $write_{\mathtt{true}}$ and $write_{\mathtt{false}}$, where $write_{\mathtt{true}}(\mathsf{h}) = \mathsf{h}'$ such that $\mathsf{h}'(\mathsf{l}) = \mathtt{true}$ and for all other locations $\mathsf{l}' \neq \mathsf{l}$, $\mathsf{h}'(\mathsf{l}') = \mathsf{h}(\mathsf{l}')$. When $\mathsf{l}_1 \neq \mathsf{l}_2$ then the induced abstract locations are independent.

The next example illustrates that abstract locations may be independent although their footprints share some concrete locations. Fix a concrete location l and define two abstract locations $\mathfrak{l}_1$ and $\mathfrak{l}_2$ both with footprint consisting of the location l. Moreover, $(\mathsf{h}, \mathsf{h}')$ belong, respectively, to the rely of location $\mathfrak{l}_i$ ($i = 1, 2$) if $\mathsf{h}(\mathsf{l})$ and $\mathsf{h}'(\mathsf{l})$ are both integers whose i-th significant bit agrees. The "guarantee" $\mathfrak{l}_i^G$ might then contain functions that set the i-th bit to some fixed value and leave the other bits alone. It is easy to see that $\mathfrak{l}_1, \mathfrak{l}_2$ are independent.

Thirdly, let $\mathsf{l}_1, \mathsf{l}_2$ be two distinct concrete locations and for heap h and finite integer sets U_1, U_2 define $P(\mathsf{h}, U_1, U_2)$ to mean that in h the locations $\mathsf{l}_1, \mathsf{l}_2$ point to non-overlapping integer lists with *sets* of elements U_1 and U_2. Now define abstract location $\mathfrak{l}_i$ by $\mathfrak{l}_i^R = \{(\mathsf{h}, \mathsf{h}') \mid \exists U_1, U_2.P(\mathsf{h}, U_1, U_2) \wedge P(\mathsf{h}', U_1, U_2)\}$ and $\mathfrak{l}_i^F(\mathsf{h}) =$ "locations reachable from l_i" if l points to a well-formed list of integers in h and $\emptyset$ otherwise. The guarantee component $\mathfrak{l}_i^G$ contains all the (idempotent) functions ι that leave the locations not in the footprint of $\mathfrak{l}_i$ alone. That $\iota(\mathsf{h}) = \mathsf{h}'$, such that $\mathsf{h}'(\mathsf{l}') = \mathsf{h}(\mathsf{l}')$ for all $\mathsf{l}' \in \mathrm{dom}(\mathsf{h}) \setminus \mathfrak{l}_i^F$. Again, $\mathfrak{l}_1$ and $\mathfrak{l}_2$ are independent.

The role of the footprints $\mathfrak{l}^F$ is to provide a minimum amount of interaction with physical allocation. If $\mathfrak{l}$ is an abstract location and h_0 the current heap so that $(\mathsf{h}_0, \mathsf{h}_0) \in \mathfrak{l}^R$ then we may, e.g., allocate $(\mathsf{h}_1, \mathsf{l}) = \mathit{new}(\mathsf{h}_0, \mathit{int}(0))$, and define an abstract location $\mathfrak{l}_1$ by

$$\begin{aligned}\mathfrak{l}_1^R &= \{(\mathsf{h}, \mathsf{h}') \mid \mathsf{h}(\mathsf{l}) = \mathsf{h}'(\mathsf{l}) \in \mathit{int}(\mathbb{Z}) \wedge \mathsf{l} \notin \mathfrak{l}^F(\mathsf{h}) \wedge \mathsf{l} \notin \mathfrak{l}^F(\mathsf{h}')\}\\ \mathfrak{l}_1^G &= \{\iota \mid \iota(\mathsf{h}) = \mathsf{h}_1 \Rightarrow \forall \mathsf{l}' \neq \mathsf{l}.\mathsf{h}(\mathsf{l}') = \mathsf{h}_1(\mathsf{l}')\}\\ \mathfrak{l}_1^F(\mathsf{h}) &= \{\mathsf{l}\}\end{aligned}$$

We now know that $\mathfrak{l}$ and $\mathfrak{l}_1$ are independent and, furthermore, $(\mathsf{h}_1, \mathsf{h}_1) \in (\mathfrak{l} \otimes \mathfrak{l}_1)^R$.

Definition 10. *Abstract locations $\mathfrak{l}_1, \ldots, \mathfrak{l}_n$ are mutually independent if they are pairwise independent and whenever $(\mathsf{h}_i, \mathsf{h}_i) \in \mathfrak{l}_i$ for $i = 1 \ldots n$ then there is h such that $(\mathsf{h}_i, \mathsf{h}) \in \mathfrak{l}_i$ for $i = 1 \ldots n$.*

Lemma 1. *Abstract locations $\mathfrak{l}_1, \ldots, \mathfrak{l}_{n+1}$ are mutually independent iff $\mathfrak{l}_1, \ldots, \mathfrak{l}_n$ are mutually independent and $\mathfrak{l}_{n+1}$ is independent of $\mathfrak{l}_1 \otimes \cdots \otimes \mathfrak{l}_n$.*

4.3 Heap PERs We are now ready to formulate the second instantiation *Heap PERs*. We assume an infinite set of *regions Regs*. A world w comprises a finite set of mutually independent abstract locations (written w) and as in the case of flat stores a tagging of locations with regions from *Regs* location. We write $\mathfrak{l} \in \mathsf{w}(\mathsf{r})$ to mean that $\mathfrak{l} \in \mathsf{w}$ is tagged with r. We define $\mathfrak{S}\mathsf{w} = \{\mathsf{h} \in \mathbb{H} \mid \forall \mathfrak{l} \in \mathsf{w}.(\mathsf{h}, \mathsf{h}) \in \mathfrak{l}^R\}$ and $\mathfrak{S}\mathsf{w}(\sigma, \sigma') = \{\star\} \iff \forall \mathfrak{l} \in \mathsf{w}.(\sigma, \sigma') \in \mathfrak{l}^R$ and $\mathfrak{S}\mathsf{w}(\sigma, \sigma') = \emptyset$ otherwise. Again, $\mathsf{h} \Vdash_{\mathsf{w}} \sigma$ iff $\mathsf{h} = \sigma$.

A morphism from w to w' is given by an injective function $u_0 : \mathsf{w} \to \mathsf{w}'$ and a pair of partial continuous functions $u_1, u_2 : \mathbb{H} \rightharpoonup \mathbb{H}$. Intuitively, the function u_1 is used to map the heaps in the PERs of locations in w to w' according to the renaming of locations specified in u_0, while u_2 does the same but from w' to w. Formally, $\forall \sigma, \sigma' \in \mathfrak{S}\mathsf{w}.\forall \mathfrak{l} \in \mathsf{w}.(\sigma, \sigma') \in \mathfrak{l}^R \Rightarrow (u_1(\sigma), u_1(\sigma')) \in u_0(\mathfrak{l})^R \wedge (u_2(u_1(\sigma)), \sigma) \in \mathfrak{l}^R$ and $\forall \sigma, \sigma' \in \mathfrak{S}\mathsf{w}'.\forall \mathfrak{l} \in \mathsf{w}.(\sigma, \sigma') \in u_0(\mathfrak{l})^R \Rightarrow (u_2(\sigma), u_2(\sigma')) \in \mathfrak{l}^R \wedge (u_1(u_2(\sigma)), \sigma) \in u_0(\mathfrak{l})^R$. The same is valid for guarantees of locations, by replacing $\cdot^R$ by $\cdot^G$. Now, $\mathfrak{S}u(\sigma) = u_2(\sigma)$. Such a morphism u is an inclusion if u_0 is an inclusion and u_1, u_2 are the identity function.

The elementary effects track reading, writing, and allocating at the level of regions: wr_{r} (writing within region r), rd_{r} (reading from within region r), al_{r} (allocating within region r). The sets of relations on $\mathfrak{S}$ modelling elementary effects are then given by

$$\begin{aligned}R \in \mathcal{R}(rd_{\mathsf{r}}) &\iff (\sigma, \sigma') \in R\mathsf{w} \Rightarrow \forall \mathfrak{l} \in \mathsf{w}(\mathsf{r}).(\sigma, \sigma') \in \mathfrak{l}^R\\ R \in \mathcal{R}(wr_{\mathsf{r}}) &\iff (\sigma, \sigma') \in R\mathsf{w} \Rightarrow \forall \mathfrak{l} \in \mathsf{w}(\mathsf{r}).\forall \iota \in \mathfrak{l}^G.(\iota(\mathsf{h}), \iota(\mathsf{h}')) \in R\mathsf{w}\\ R \in \mathcal{R}(al_{\mathsf{r}}) &\iff (\sigma, \sigma') \in R\mathsf{w} \Rightarrow \forall \mathsf{w}_1.\forall u \in \mathbf{I}(\mathsf{w}, \mathsf{w}_1).(\mathsf{w}_1 \setminus \mathsf{w} \subseteq \mathsf{w}_1(\mathsf{r})) \Rightarrow \forall \sigma_1, \sigma_1' \in \mathfrak{S}\mathsf{w}_1.\\ &\qquad (\sigma_1.u \sim \sigma \wedge \sigma_1'.u \sim \sigma' \wedge (\sigma_1, \sigma_1') \in \textstyle\bigcap_{\mathfrak{l} \in \mathsf{w}_1 \setminus \mathsf{w}} \mathfrak{l}^R) \Rightarrow (\sigma_1, \sigma_1') \in R\mathsf{w}_1\end{aligned}$$

Thus, a relation $R \in \mathcal{R}(rd_{\mathsf{r}})$ ensures that locations being read contain "equal" (in the sense of $\mathfrak{l}^R$) values; a relation $R \in \mathcal{R}(wr_{\mathsf{r}})$ is oblivious to 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 it only adds 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.

5 Proof-relevant Logical Relations

Given an instantiation, e.g. one of the above examples, we interpret types (and typing contexts) as p.p.f. over $\mathbf{W}$ and types with effect as a fibred setoid over $S(\mathbf{W})$. A term in context $\Gamma \vdash e : \tau \;\&\; \varepsilon$ will be interpreted as a morphism $[\![e]\!]$ from $S([\![\Gamma]\!])$ to $T_\varepsilon[\![\tau]\!]$ where T_ε takes p.p.f. and effects to fibred setoids and is given below in Definition 13.

Derivations of equations will be interpreted as equality proofs between the corresponding morphisms and can be used to deduce observational equivalences (Theorem 3).

This, however, requires a loose relationship of the setoid interpretation with the actual meanings of raw terms which is given by realization relations $\Vdash^A$. Their precise format and role are described in the following two definitions.

Definition 11. *A* semantic type *is a pair $(A, \Vdash^A)$ where A is a p.p.f. (on $\mathbf{W}$) and $\Vdash^A_{\mathsf{w}}$ is an admissible subset of $\mathbb{V} \times A\mathsf{w}$ for each $\mathsf{w} \in \mathbf{W}$ such that for every inclusion $u : \mathsf{w} \to \mathsf{w}'$ one has that $v \Vdash^A_{\mathsf{w}} \mathsf{v}$ implies $v \Vdash^A_{\mathsf{w}'} u.\mathsf{v}$. A* semantic computation *is a pair $(T, \Vdash^T)$ where T is a fibred setoid over $\mathbf{W}$ and $\Vdash^T_{\mathsf{w}}$ is an admissible subset of $\mathbb{C} \times T\mathsf{w}$ for each w.*

Definition 12. *Let $(\Gamma, \Vdash^\Gamma)$ and $(A, \Vdash^A)$ be semantic types and let $(T, \Vdash^T)$ be a semantic computation. If $e : S(\Gamma) \to T$ is a morphism of fibred setoids and $f : \mathbb{V} \to \mathbb{C}$ then we write $f \Vdash^{\Gamma \vdash T} e$ to mean that for some representative (f_0, f_1) of f one has that whenever $\eta \Vdash^\Gamma_{\mathsf{w}} \gamma$ then $f_0(\eta) \Vdash^T_{\mathsf{w}} e(\gamma)$ holds for all worlds w.*

The following definition, corresponding to that in Fig. 1, is where the machinery introduced above pays off. In particular, it defines the semantics of computations, where proofs, i.e., pullback squares, are constructed.

Definition 13. *Let A be a semantic type and ε an effect. A semantic computation $T_\varepsilon A$ is defined as follows:*

- *(Objects) Elements of $(T_\varepsilon A)\mathsf{w}$ are pairs $(\mathsf{c}_0, \mathsf{c}_1)$ of partial continuous functions where*
  $\mathsf{c}_0 : \Im\mathsf{w} \rightharpoonup \Sigma\mathsf{w}_1.\mathbf{I}(\mathsf{w}, \mathsf{w}_1) \times \Im\mathsf{w}_1 \times A\mathsf{w}_1$

and c_1 is as follows. If $R \in \mathcal{R}(\varepsilon)$ and $(\sigma, \sigma') \in R\mathsf{w}$ then $\mathsf{c}_1(R, \sigma, \sigma')$ either is undefined and $\mathsf{c}_0(\sigma)$ and $\mathsf{c}_0(\sigma')$ are both undefined or else $c_1(R, \sigma, \sigma')$ is defined and then $\mathsf{c}_0(\sigma)$ and $\mathsf{c}'_0(\sigma')$ are both defined, say $\mathsf{c}_0(\sigma) = (\mathsf{w}_1, u, \sigma_1, a)$ and $\mathsf{c}_0(\sigma') = (\mathsf{w}'_1, u', \sigma'_1, a')$. In this case, $\mathsf{c}_1(R, \sigma, \sigma')$ 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, $p \in A\overline{w}(x.a, x'.a')$ and, finally, $(\sigma_1.u, \sigma'_1.u') \in R\underline{\mathsf{w}}$ where $\underline{\mathsf{w}}$ and $\overline{\mathsf{w}}$ are low point and apex of ${}^x_v\Diamond^{x'}_{v'}$.

- *(Proofs) As usual, proofs only look at the $(-)_0$ components. Thus, if $(\mathsf{c}_0, \_) \in T_\varepsilon A\mathsf{w}$ and $(\mathsf{c}'_0, \_) \in T_\varepsilon A\mathsf{w}'$ and ${}^x_v\Diamond^{x'}_{v'}$ is in $S(\mathbf{W})(\mathsf{w}, \mathsf{w}')$ with apex and low point $\overline{\mathsf{w}}, \underline{\mathsf{w}}$ then a proof in $(T_\varepsilon A)^x_v\Diamond^{x'}_{v'}(\mathsf{c}, \mathsf{c}')$ is a partial continuous function μ which given $\sigma \in \Im\mathsf{w}$ and $\sigma' \in \Im\mathsf{w}'$ and $p : \sigma.v \sim \sigma'.v'$ either is undefined and then $\mathsf{c}_0(\sigma)$ and $\mathsf{c}'_0(\sigma')$ are both undefined or else is defined and then $\mathsf{c}_0(\sigma)$ and $\mathsf{c}'_0(\sigma')$ are both defined with results, say, $\mathsf{c}_0(\sigma) = (\mathsf{w}_1, u, \sigma_1, \mathsf{v})$ and $\mathsf{c}'_0(\sigma') = (\mathsf{w}'_1, u', \sigma'_1, \mathsf{v}')$. In this case, $\mu(p)$ returns a tuple $({}^{x_1}_{v_1}\Diamond^{x'_1}_{v'_1}, q)$ satisfying $x_1uv = x'_1u'v'$ and $q \in A\overline{\mathsf{w}_1}(x_1.\mathsf{v}, x'_1.\mathsf{v}')$ with $\overline{\mathsf{w}_1} = \mathrm{cod}(x_1)$ and $\sigma_1.v_1 \sim \sigma_1.v'_1$ in $\Im\underline{\mathsf{w}_1}$.*

- *(Realization) If $c \in \mathbb{C}$, we define $c \Vdash^{T_\varepsilon A}_{\mathsf{w}} (\mathsf{c}_0, \mathsf{c}_1)$ to mean that whenever $\mathsf{h} \Vdash_{\mathsf{w}} \sigma$ then $c(\mathsf{h})$ is defined iff $\mathsf{c}_0(\sigma)$ is defined and if $c(\mathsf{h}) = (\mathsf{h}_1, v)$ and $\mathsf{c}_0(\sigma) = (\mathsf{w}_1, u, \sigma_1, \mathsf{v})$ then $\mathsf{h}_1 \Vdash_{\mathsf{w}_1} \sigma_1$ and $v \Vdash^A_{\mathsf{w}_1} \mathsf{v}$.*

Proving that a semantic computation $T_\varepsilon A$ as in Definition 13 is a fibred setoid is non-trivial. The tricky case is the existence of a transitivity operation. It is here that we need the independence of abstract locations as stated in Definition 9, which implies that $\Im$ is also minimal-pullback-preserving. Details, along with the construction of the cartesian product $(A \times B, \Vdash^{A \times B})$ and function space $(A \Rightarrow T, \Vdash^{A \Rightarrow T})$, given semantic types $(A, \Vdash^A)$ and $(B, \Vdash^B)$ and computation $(T, \Vdash^T)$, may be found in the appendix.

5.1 Fundamental theorem Given a semantic type $[\![A]\!]$ for each basic type A we can interpret any type τ as a semantic type $[\![\tau]\!]$ by putting $[\![\tau_1 \xrightarrow{\varepsilon} \tau_2]\!] = [\![\tau_1]\!] \Rightarrow T_\varepsilon [\![\tau_2]\!]$. A typing context $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ is interpreted as the semantic type $[\![\Gamma]\!] = (1 \times [\![\tau_1]\!]) \times \ldots) \times [\![\tau_n]\!]$ where 1 is the constant functor returning the discrete setoid $\{()\}$.

To every typing derivation $\Gamma \vdash t : \tau \;\&\; \varepsilon$ we then associate a morphism $[\![\Gamma \vdash t : \tau \;\&\; \varepsilon]\!] : S([\![\Gamma]\!]) \to T_\varepsilon [\![\tau]\!]$ such that $[\![t]\!] \Vdash^{[\![\Gamma]\!] \to T_\varepsilon \tau} [\![\Gamma \vdash t : \tau \;\&\; \varepsilon]\!]$. (Note: *this* is point where the untyped semantics is related with the abstract one.) For every equality derivation $\Gamma \vdash t = t' : \tau \;\&\; \varepsilon$ we have $[\![\Gamma \vdash t : \tau \;\&\; \varepsilon]\!] = [\![\Gamma \vdash t' : \tau \;\&\; \varepsilon]\!]$, where the two typing derivations $\Gamma \vdash t : \tau \;\&\; \varepsilon$ and $\Gamma \vdash t' : \tau \;\&\; \varepsilon$ are the canonical ones associated with the equality derivation $\Gamma \vdash t = t' : \tau \;\&\; \varepsilon$. In essence, one has to provide a semantic counterpart for every syntactic concept, e.g. let, fix, etc. Details are in the appendix.

5.2 Observational equivalence Let Int stand for the constant functor that returns the discrete setoid on the set $\mathbb{Z}$ of integers. We define $v \Vdash^{\mathrm{Int}}_{\mathsf{w}} i \iff v = int(i)$. We also assume that there is some initial store and abstract store h_0, σ_0 and a world w_0 such that $\mathsf{h}_0 \Vdash_{\mathsf{w}_0} \sigma_0$. For instance, w_0 can be the empty world with no locations and accordingly h_0 the initial store at startup.

Definition 14. *Let $(A, \Vdash^A)$ be a semantic type. We define an* observation of type A *as a morphism $o : A \to T_\varepsilon \mathrm{Int}$ for some ε and a function f so that $f \Vdash^{A \to T_\varepsilon \mathrm{Int}} o$.*

Two values v, v' are observationally equivalent at type A *if for all observations f, o of type A one has that $f(v)(\mathsf{h}_0)$ is defined iff $f(v')(\mathsf{h}_0)$ is defined and when $f(v)(\mathsf{h}_0) = (\mathsf{h}_1, v_1)$ and $f(v')(\mathsf{h}_0) = (\mathsf{h}'_1, v'_1)$ then $v_1 = v'_1$.*

Taking $o = [\![\vdash f : \tau \xrightarrow{\varepsilon} \mathtt{int}]\!]$ immediately yields the following:

Proposition 2. *If v, v' are observationally equivalent at type $[\![\tau]\!]$ and f is a term such that $\vdash f : \tau \xrightarrow{\varepsilon} \mathtt{int}$ then $[\![f]\!](v)(\mathsf{h}_0)$ is defined iff $[\![f]\!](v')(\mathsf{h}_0)$ is defined and when $[\![f]\!](v)(\mathsf{h}_0) = (\mathsf{h}_1, v_1)$ and $[\![f]\!](v')(\mathsf{h}_0) = (\mathsf{h}'_1, v'_1)$ then $v_1 = v'_1$.*

Theorem 3 (Observational equivalence). *If $(A, \Vdash^A)$ is a semantic type and $v \Vdash^A_{\mathsf{w}_0} e$ and $v' \Vdash^A_{\mathsf{w}_0} e'$ with $e \sim e'$ in A_{w_0} then v and v' are observationally equivalent at type A.*

Proof We have $f(v) \Vdash^{T_\varepsilon \mathrm{Int}}_{\mathsf{w}_0} o(e)$ and $f(v') \Vdash^{T_\varepsilon \mathrm{Int}}_{\mathsf{w}_0} o(e')$ and also $\mu : o(e) \sim_{{}^1_1 \Diamond^1_1} o(e')$ in $T_\varepsilon \mathrm{Int}$ for some μ as in Definition 13.

The application μ to $\sigma_0, \sigma_0, r(\sigma_0)$ either is undefined in which case $o(e)(\sigma_0)$ and $o(e')(\sigma_0)$ and $f(v)(\mathsf{h}_0)$ and $f(v')(\mathsf{h}_0)$ are all undefined, the latter by the definition of $\Vdash^{T_\varepsilon \mathrm{Int}}$. Otherwise, we get $f(v)(\mathsf{h}_0) = (\mathsf{h}_1, v_1)$ and $f(v')(\mathsf{h}_0) = (\mathsf{h}'_1, v'_1)$ and $o(e)(\sigma_0) = (\sigma_1, i_1)$ and $o(e')(\sigma_0) = (\sigma'_1, i'_1)$ where, by definition of realization in $T_\varepsilon \mathrm{Int}$ and Int, we have $v_1 = int(i_1)$ and $v_2 = int(i_2)$. Now, $\mu(\sigma_0, \sigma_0, r(\sigma_0))$ returns a pullback $({}^{x_1}_{v_1} \Diamond^{x'_1}_{v'_1}, q)$ such that, in particular, $x_1.i_1 \sim x_2.i_2$, whence $i_1 = i_2$ since Int is constant and then $v_1 = v_2$ as required. □

6 Applications

In what follows we use our semantics to establish a number of effect-dependent semantic equalities, hence program equivalences in the sense of observational equivalences. We also give some semantically justified typings of concretely given functions, in particular "set factory" described in Section 2.1. More examples are discussed in the appendix.

6.1 Sets of locations We work in the instantiation "sets of locations". Recall the example, "dummy allocation" from Section 2.1. Suppose that $f \Vdash^{\Gamma \vdash T_\varepsilon A} e$. Now, put $dummy(e)(\mathsf{w})(\gamma \in [\![\Gamma]\!]\mathsf{w})(\mathsf{h} \in \mathfrak{S}\mathsf{w}) = e(\mathsf{w})(\gamma)(\mathsf{h}')$, where h' is the heap obtained by adding a dummy location to h. We have $dummy(f) \Vdash^{\Gamma \vdash T_\varepsilon A} dummy(e)$ since $\Vdash$ is oblivious to extensions of the store. Therefore, reflexivity also furnishes a proof of equality. It also means that, semantically, $dummy(f)$ does not need to flag the allocation effect *al* since no semantically visible world extension takes place.

For the Interleaved Dummy Allocation example, on the other hand, there is an extra step caused by the proper allocation, which yields a world extension $\mathsf{w} \to \mathsf{w}_1$ and $\mathsf{w} \to \mathsf{w}'_1$. In order to show the equivalence, we construct a proof, i.e., a pull-back square $\mathsf{w}_1 \diamond \mathsf{w}'_1$, where the allocated concrete locations are identified in its low point. Then the reasoning is the same as above used for showing the semantic equivalence of the Dummy example.

This is different in the following example. Define a semantic type N of names by letting $N\mathsf{w}$ be the discrete setoid on the set w and $Nu(\mathsf{l}) = u(\mathsf{l})$ and $v \Vdash^N_{\mathsf{w}} \mathsf{l} \iff v = loc(\mathsf{l})$. Moreover, $f = [\![\mathtt{ref}(0)]\!], g = [\![\mathtt{let}\ x \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ (x, y)]\!]$, and $h = [\![\mathtt{let}\ x \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ (y, x)]\!]$. We now define semantic counterparts $\mathsf{f} : S(1) \to T_{al}N$, $\mathsf{g}, \mathsf{h} : S(1) \to T_{al}N$, where

$$\mathsf{f}_0\mathsf{w}(\sigma) = (\mathsf{w}_1, i_1, \sigma_1, \mathsf{l}_1),\ \mathsf{g}_0\mathsf{w}(\sigma) = (\mathsf{w}_2, i_2 i_1, \sigma_2, (\mathsf{l}_1, \mathsf{l}_2)),\ \text{and}\ \mathsf{h}_0\mathsf{w}(\sigma) = (\mathsf{w}_2, i_2 i_1, \sigma_2, (\mathsf{l}_2, \mathsf{l}_1))$$

Here and in what follows it is assumed that $new(\sigma) = (\mathsf{l}_1, \sigma_1)$ and $new(\sigma_1) = (\mathsf{l}_2, \sigma_2)$ and $\mathsf{w}_1 = \mathsf{w} \cup \{\mathsf{l}_1\}$ and $\mathsf{w}_2 = \mathsf{w}_1 \cup \{\mathsf{l}_2\}$. Recall that $\mathfrak{S}\mathsf{w} \subseteq \mathbb{H}$. Finally, $i_1 : \mathsf{w} \to \mathsf{w}_1$ and $i_2 : \mathsf{w}_1 \to \mathsf{w}_2$ stand for the obvious inclusions. We use analogous definitions for the primed variants.

In order to define $\mathsf{f}_{0.5}$ we start with $u : \mathsf{w} \to \mathsf{w}'$ and $\sigma \in \mathfrak{S}\mathsf{w}, \sigma' \in \mathfrak{S}\mathsf{w}'$, $R \in \mathcal{R}(al)$ such that $(\sigma, u.\sigma') \in R\mathsf{w}$. Define $u' : \mathsf{w}_1 \to \mathsf{w}'_1$ so that $u' i_1 = i'_1 u$, that is $u'(\mathsf{l} \in \mathsf{w}) = u(\mathsf{l})$, $u'(\mathsf{l}_1) = \mathsf{l}'_1$. We now return the pullback square $\mathsf{w}_1 {}^{u'}_{1}\diamond^{1}_{u'} \mathsf{w}'_1$ with apex w'_1 and low point w_1 and the trivial proof that $u'.\mathsf{l}_1 = \mathsf{l}'_1$. This settles the definition of $\mathsf{f}_{0.5}$, since $R\mathsf{w}_1$ is total since $R \in \mathcal{R}(al)$. Notice though, that we cannot avoid the allocation effect here.

The functions $g_{0.5}$ and $h_{0.5}$ are defined analogously.

We now construct a proof that $\mathsf{g} \sim \mathsf{h}$, recall that only g_0 and h_0 are needed for this. Given w, σ and the notation from above this proof amounts to a pullback square $\mathsf{w}_2 {}^{x}_{v}\diamond^{x'}_{v'} \mathsf{w}'_2$ such that $x i_2 i_1 = x' i'_2 i'_1 u$ and $x.(\mathsf{l}_1, \mathsf{l}_2) = x'.(\mathsf{l}_2, \mathsf{l}_1)$ and $\sigma_2.v \sim \sigma'_2.v'$. Note that, accidentally, the final abstract stores of both computations are the same, namely, σ_2. Now let f be the bijection that swaps $\mathsf{l}_1, \mathsf{l}_2$ and fixes everything else. We then put ${}^{x}_{v}\diamond^{x'}_{v'} := {}^{1}_{f}\diamond^{f}_{1}$. Now, obviously $(\mathsf{l}_1, \mathsf{l}_2) = f.(\mathsf{l}_2, \mathsf{l}_1)$ and $\sim$-equality of abstract stores is trivial by definition.

6.2 Heap PERs In this section we generalize our earlier collection of effect-dependent program equivalences [4] to the abstract locations of the Heap PERs instantiation. We first show how the set factory indeed has the announced effect typings and thus can participate in effect-dependent equivalences.

Set factory Let w be a world and $\sigma \in \mathfrak{S}\mathsf{w}$. Suppose that σ_1 arises from σ by allocating a fresh set data structure, e.g., a linked list, with entry point(s) E. Let l_1 be the abstract location describing this fresh data structure, i.e., $(\mathsf{h}, \mathsf{h}') \in \mathsf{l}^R_1 \iff$ the data structures starting from E in h, h' are well-formed, denote the same set, and do not overlap with the footprints of all the abstract locations in w. The footprint l^F_1 comprises the locations that make up this data structure assuming that $(\mathsf{h}, \mathsf{h}) \in \mathsf{l}^R$, otherwise any value can be

chosen. Finally, $\mathfrak{l}^G$ contains idempotent functions, ι, such that $\iota(\mathsf{h}) = \mathsf{h}_1$ and h_1 agree on all concrete locations from $\mathrm{dom}(\mathsf{h}) \supseteq \mathfrak{l}^F(\mathsf{h})$ and, moreover, $\mathrm{dom}(\mathsf{h}_1) \supseteq \mathrm{dom}(\mathsf{h})$.

Now for any chosen region r we add $\mathfrak{l}_1$ to r to yield a new world w_1. The function $\mathit{setfactory}_0\mathsf{w}\sigma$ 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 $\sigma_1 \in \mathfrak{S}\mathsf{w}_1$ and $i \in \mathbb{Z}$ then the reading function looks up i in the data structure starting at the entry points E in σ_1. (Note that $\sigma_1 \in \mathfrak{S}\mathsf{w}$ asserts that this data structure exists and is well-formed.) The returned (abstract) store σ_2 might not be the same as σ because internal reorganizations, e.g., removal of duplicates, might have occurred. However, no world extension is needed and $\sigma_1 \sim \sigma_2$ holds. This together with the fact that the outcome only depends on the $\mathfrak{l}^R$ equivalence class justifies a read-only typing for reading.

Memoization For the simple *memo* functional from Section 2.1 we produce just as in the previous example a fresh abstract location $\mathfrak{l}$ that contains the two newly allocated concrete locations, say $\mathfrak{l}_x, \mathfrak{l}_y$, and on which we impose the invariant $(\mathsf{h}, \mathsf{h}') \in \mathfrak{l}^R \iff \mathsf{h}(\mathfrak{l}_x), \mathsf{h}'(\mathfrak{l}_x)$ contain the same integer value, say i and that $\mathsf{h}(\mathfrak{l}_y), \mathsf{h}'(\mathfrak{l}_y)$ both contain the integer value $f(i)$ where f is the pure function to be memoised.

Effect-dependent equivalences Consider the following notation

$$\sigma \sim_{\mathrm{rds}(\varepsilon,\mathsf{w})} \sigma' \iff \forall \mathfrak{l} \in \mathsf{w}(\mathrm{rds}(\varepsilon)).(\sigma, \sigma') \in \mathfrak{l}^R$$
$$\sigma \sim_{\mathit{nwrs}(\varepsilon,\mathsf{w})} \sigma' \iff \forall \mathfrak{l} \in \mathsf{w}(\mathit{nwrs}(\varepsilon)).(\sigma, \sigma') \in \mathfrak{l}^R$$

which specify that the abstract heaps σ and σ' are equivalent on all the abstract locations $\mathfrak{l}$ in regions associated, respectively, to read effects and no-writes in ε.

Lemma 2. *Let $\Gamma \vdash e : \tau \;\&\; \varepsilon$. For any world $\mathsf{w} \in \mathbf{W}$, and context $\gamma \in [\![\Gamma]\!]\mathsf{w}$, whenever $\sigma_0, \sigma_0' \in \mathfrak{S}\mathsf{w}$ such that $\sigma_0 \sim_{\mathrm{rds}(\varepsilon,\mathsf{w})} \sigma_0'$, then $\mathsf{c}(\sigma_0)$ and $\mathsf{c}(\sigma_0')$ where $\mathsf{c} = [\![\Gamma \vdash e : \tau \;\&\; \varepsilon]\!]\mathsf{w}(\gamma)$ are equally defined and if $\mathsf{c}(\sigma_0) = (\mathsf{w}_1, u, \sigma_1, \mathsf{v})$ and $\mathsf{c}(\sigma_0') = (\mathsf{w}_1', u', \sigma_1', \mathsf{v}')$ then there exist (continuously!) a pullback $\mathsf{w}_1 \,{}^{x}_{y}\!\diamond^{x'}_{y'}\, \mathsf{w}_1'$ with apex $\overline{\mathsf{w}}$ and low point $\underline{\mathsf{w}}$ and a proof of $x.\mathsf{v} \sim x'.\mathsf{v}'$ such that $xu = x'u'$ and the following is satisfied:*

1. *for all $\mathfrak{l} \in \mathsf{w}$, we have either: $(\sigma_0, \sigma_1.u) \in \mathfrak{l}^R$ and $(\sigma_0', \sigma_1'.u') \in \mathfrak{l}^R$ (remain equivalent) or $(\sigma_1.u, \sigma_1'.u') \in \mathfrak{l}^R$ (equally modified);*
2. *if $\mathfrak{l} \in \mathsf{w}(\mathit{nwrs}(\varepsilon))$, then $(\sigma_0, \sigma_1.u) \in \mathfrak{l}^R$ and $(\sigma_0', \sigma_1'.u') \in \mathfrak{l}^R$.*
3. *There exists a morphism $\mathsf{c}' \in [\![\Gamma]\!] \to T_\varepsilon[\![\tau]\!]$, such that $\mathsf{c}' \sim \mathsf{c}$ and if $\mathsf{c}'(\mathsf{w})(\gamma)\sigma_0 = (\mathsf{w}_\star, u_\star, \sigma_\star, \mathsf{v}_\star)$, then for all regions $\mathsf{r} \notin \mathrm{als}(\varepsilon)$, $\mathsf{w}_\star(\mathsf{r}) = \mathsf{w}(\mathsf{r})$.*

We can validate all the effect-dependent program equivalences "dead, commuting, duplicated computation" and "pure lambda hoist", as well as the "masking rule" from previous work [6] in our new, more powerful, setting. To give an impression of the formulation of these validations we state the corresponding proposition for "dead computation" which is particularly interesting in that it contains a termination precondition. The proof, and details of the other equations are in the appendix, which also contains a validation of loop unrolling optimisation described by Tristan and Leroy [31].

Proposition 3 (dead computation). *Suppose that $\Gamma \vdash e : \mathtt{unit} \;\&\; \varepsilon$, that $\mathrm{wrs}(\varepsilon) = \emptyset$ and that $[\![\Gamma \vdash e : \mathtt{unit} \;\&\; \varepsilon]\!]\mathsf{w}(\gamma)(\sigma)$ is defined for all $\mathsf{w}, \gamma \in [\![\Gamma]\!]\mathsf{w}, \sigma \in \mathfrak{S}\mathsf{w}$. Then if for all worlds w, all contexts $\gamma \in [\![\Gamma]\!]\mathsf{w}$, and abstract heaps $\sigma \in \mathfrak{S}\mathsf{w}$, the function $[\![\Gamma \vdash e]\!](\mathsf{w})(\gamma)(\sigma)$ is defined, then $[\![\Gamma \vdash e : \mathtt{unit} \;\&\; \varepsilon]\!] \sim [\![\Gamma \vdash () : \mathtt{unit} \;\&\; \varepsilon]\!]$.*

6.3 State Dependent Abstract Data Types (ADT) We prove the equivalence of a number of programs involving state dependent abstract data types.

Awkward Example The first example is Pitts and Stark's classic *awkward* example[26]. Consider the following two programs:

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

Intuitively, the expressions 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 of these functions as follows: Assign the region where x is allocated as r. If f has the type $\texttt{unit} \xrightarrow{\varepsilon} \texttt{unit}$ with effects ε, then e_1 has type $(\texttt{unit} \xrightarrow{\varepsilon} \texttt{unit}) \xrightarrow{\varepsilon, rd_{\mathsf{r}}, wr_{\mathsf{r}}} \texttt{int}\ \&\ \varepsilon, al_{\mathsf{r}}$, while e_2 has type $(\texttt{unit} \xrightarrow{\varepsilon} \texttt{unit}) \xrightarrow{\varepsilon} \texttt{int}\ \&\ \varepsilon$. Notice that ε may contain rd_{r} or wr_{r} or both. Moreover, assume that the footprint of a location in region r consists of a single concrete location l, and that the guarantee of a location l^G consist of a single function $write_1$ such that $write_1(\mathsf{h}) = \mathsf{h}'$ where $\mathsf{h}'(\mathsf{l}) = 1$ and $\mathsf{h}'(\mathsf{l}') = \mathsf{h}(\mathsf{l}')$ for all other locations. Clearly e_1 has such a write effect.

For proving the equivalence of e_1 and e_2, assume a world w and an abstract heap σ. Let $[\![e_1]\!]\mathsf{w}\sigma = (\mathsf{w} \uplus \mathsf{w}_1 \uplus \mathsf{w}_r, u_1, \mathsf{v}_1, \sigma_1)$ and $[\![e_2]\!]\mathsf{w}\sigma = (\mathsf{w} \uplus \mathsf{w}_1, u_1, \mathsf{v}_2, \sigma_2)$. We need to 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 v_1 and v_2 are equal in its apex and σ_1 and σ_2 are equal in its low point. Since wr_{r} is in the effects of e_1, we have that $\mathsf{v}_1 = 1$. We also have $\mathsf{v}_2 = 1$ trivially. Hence v_1 and v_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, σ_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 σ_2.

Modified Awkward Example Consider now the following variant of the Awkward example, due to Dreyer et al.[14]:

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

The difference is that in the first program x is written to 0 and the call-back function is used twice. Interestingly, however, the solution given for the Awkward example 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 abstract location assigned for x is to write one. Therefore, if f has effect of writing on the region r, it will set x to one.

Callback with Lock Example We now show equivalence of the following programs, also due to Dreyer et al.[14]:

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

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

Both programs produce a pair of functions, one incrementing the value stored in x and the second returning the value stored in x. The boolean reference b serves as lock in the incrementing function. Once this function is called the value in b is set to `false` and only after calling the call-back, the value in x is incremented is b set again to `true`. However, the implementation of the increment function is different. While the program to the left calls the call-back function $f()$ and then increments the value of x using the value stored in x, the program to the right remembers (in n) the value of x before the call-back is called and then uses it to increment the value stored in x.

Assume that x and b are in the footprint of the same abstract location (l) in the region r. We show that these programs are equivalent under the type

$(\mathtt{unit} \xrightarrow{\varepsilon} \mathtt{unit}) \xrightarrow{\varepsilon, wr_{\mathsf{r}}, rd_{\mathsf{r}}} \mathtt{unit}) \times (\mathtt{unit} \xrightarrow{rd_{\mathsf{r}}} \mathtt{unit})$ & $al_{\mathsf{r}}, \varepsilon$,
where ε may contain the effects $wr_{\mathsf{r}}, rd_{\mathsf{r}}$. In particular, the location l is specified as follows: its footprint consists only of the concrete locations storing x and b, written l_b and l_x, while its rely-condition is equality. The more interesting is its guarantee condition (l^G), which contains the following idempotent functions f_i for $i \in \mathbb{N}$: $f_i(\mathsf{h}) = \mathsf{h}$ if $\mathsf{h}(\mathsf{l}_b) = \mathtt{false}$ and $f_i(\mathsf{h}) = \mathsf{h}'$ if $\mathsf{h}(\mathsf{l}_b) = \mathtt{true}$, where $\mathsf{h}'(\mathsf{l}_x) = i$ if $\mathsf{h}(\mathsf{l}_x) \leq i$ and $\mathsf{h}'(\mathsf{l}_x) = \mathsf{h}(\mathsf{l}_x)$; moreover, the value of b is unchanged, that is, $\mathsf{h}'(\mathsf{l}_b) = \mathsf{h}(\mathsf{l}_b)$. It is easy to check that these functions are idempotent as well as their composition.

First, notice that indeed the two functions above have type wr_{r} as the increment of x is captured by using some write function f_i and moreover b is `true`. Now, to show that the two programs above are equivalent, we need to show that the value stored in x before and after the call back is called is the same. This is the case, as even if $wr_{\mathsf{r}} \in \varepsilon$, the value stored in b is `false`, which means that any function f_i used will leave the concrete locations storing x and b untouched.

Notice that if the read function also called the call-back, then the reasoning above would break, as the call-back could modify the value stored in x because b is `true`.

7 Conclusions

We have laid out the basic theory of proof-relevant logical relations and shown how they can be used to justify nontrivial effect-dependent program equivalences. We have also shown that proof-relevant logical relations give direct-style justifications of the Pitts-Stark-Shinwell equivalences for name generation. 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 [19, 5, 4, 6, 30] do not provide such possibilities.

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* [11] from "design by contract" [23] and more generally data refinement [25]. 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.

Our abstract locations draw upon several ideas from separation logic [28], in particular footprints and the conditions on rely/guarantee assumptions from [32]. Intriguingly, we did not need something resembling the "frame rule" although perhaps the Π-quantification over larger worlds in function spaces plays its role.

Pullback-preserving functors and especially the instantiation *sets of locations* are inspired by FM-sets [15] or rather the *Schanuel topos* to which they are equivalent (see Staton [29] for a comprehensive account). The instantiations other than *sets of locations*, as well as the use of setoids for the "values" of these functors rather than plain sets is original to this work.

We would like to have a semi-formal format that allows one to integrate semantic arguments with typing and equality derivations more smoothly. We would also like to allow proof-relevant partial equivalences in the Heap PER instantiation, which essentially amounts to the ability to store values with proof-relevant equality. In particular, this would allow us to model higher-order store with some layering policy [9]. For unrestricted higher-order store as in [30], but with abstract locations, one would need to overcome the well-known difficulties with circular definition of worlds. Step-indexing [2] is an option, but we would prefer a domain-theoretic solution. The formal similarity of our abstract locations with the rely-guarantee formalism [12, 32] suggests the intriguing possibility of an extension to concurrency.

We also believe that update operations governed by finite state machines [1] can be modelled as an instance of our framework and thus combined with effect-dependency. The application of our general framework to effects other than reading, writing, allocation deserves further investigation.

Indeed, we feel that with the transition to proof-relevance we have opened a door to a whole new world that hopefully others will investigate with us.

References

1. A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.
2. A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, volume 3924 of *LNCS*, 2006.
3. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *J. Funct. Program.*, 13(2):261–293, 2003.
4. N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*, 2007.
5. N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*, 2009.
6. N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In *APLAS*, volume 4279 of *LNCS*, 2006.
7. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *TLCA*, volume 3461 of *LNCS*, 2005.
8. L. Birkedal, A. Carboni, G. Rosolini, and D. S. Scott. Type theory via exact categories. In *LICS*, pages 188–198. IEEE Computer Society, 1998.
9. G. Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208(6), 2010.
10. A. Carboni, P. J. Freyd, and A. Scedrov. A categorical approach to realizability and polymorphic types. In *Proc. MFPS, Springer LNCS 298*, pages 23–42, 1987.
11. 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.
12. J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.
13. T. Dinsdale-Young, P. Gardner, and M. J. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, volume 6217 of *LNCS*, 2010.
14. 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.
15. M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.
16. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, 1986.

17. M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS*, 1994.
18. J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, volume 7211 of *LNCS*, 2012.
19. O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL*, 2012.
20. N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.
21. N. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP*, 2012.
22. R. Ley-Wild and A. Nanevski. Subjective concurrent separation logic. submitted for publication, Jan. 2012.
23. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
24. E. Moggi. Notions of computation and monads. *Information and Computation*, 9(1):55–92, 1991.
25. W. P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*. Cambridge University Press, 1998.
26. A. Pitts and I. Stark. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*, 1998.
27. 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.
28. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
29. S. Staton. *Name-Passing Process Calculi: Operational Models and Structural Operational Semantics*. PhD thesis, Univ. Cambridge, 2007.
30. J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In *ICFP*, 2011.
31. J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.
32. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*. 2007.
33. V. Voevodsky. Univalent semantics of constructive type theories. In *CPP*. 2011.

A Online Appendix

This appendix contains some additional technical material that was omitted from the main body for space reasons. In particular, Section A.1 contains standard details on semantics of values and computations as well as of domain theory. Section A.2 elaborates the Setoids theory, introducing the definition of Isomorphic pullbacks and contains more properties of p.p.f. In Section A.3, a third instantiation, more complex than the sets of locations, but simpler than Heap PERs can be found. Section A.4 contains most of the machinery necessary to establish the Fundamental Theorem. Finally, Section A.5 contains further applications of our setting. For instance, we prove the soundness of a number of re-writes, such as the commuting equation, duplication elimination, pure lambda-hoist, etc. We also prove the soundness of the Masking rule and discuss the loop-unrolling example in [31].

A.1 Syntax and Semantics

Predomains 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 ascending 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 $f : A \rightharpoonup B$.

Semantics The untyped semantics of values and computations is given by the recursive clauses in Figure 5; note the overloading of semantic brackets for constants, values and computations. The notation $\eta(x)$ stands for the i-th projection from $\eta \in \mathbb{V}$ if x is x_i and $\eta[x \mapsto v]$ (functionally) updates the i-th slot in η when $x = x_i$.

$$
\begin{aligned}
[\![x]\!]\eta &= \eta(x) \\
[\![c]\!]\eta &= [\![c]\!] \\
[\![(v_1, v_2)]\!]\eta &= ([\![v_1]\!]\eta, [\![v_2]\!]\eta) \\
[\![v.i]\!]\eta &= d_i \text{ if } i = 1, 2,\ [\![v]\!]\eta = (d_1, d_2) \\
[\![\texttt{rec}\ f\ x = t]\!]\eta &= \mathit{fun}(g^\dagger\ \eta), \text{ where } g(\eta, u) = \lambda d.[\![t]\!]\eta[f \mapsto \mathit{fun}(u), x \mapsto d]
\end{aligned}
$$

$$
\begin{aligned}
[\![v]\!]\eta\ \mathsf{h} &= (\mathsf{h}, [\![v]\!]\eta) \\
[\![\texttt{if}\ v\ \texttt{then}\ t_2\ \texttt{else}\ t_3]\!]\eta\mathsf{h} &= [\![t_2]\!]\eta\mathsf{h} && \text{if } [\![v]\!]\eta = \mathit{int}(z),\ z \neq 0 \\
[\![\texttt{if}\ x\ \texttt{then}\ t_2\ \texttt{else}\ t_3]\!]\eta &= [\![t_3]\!]\eta\mathsf{h} && \text{if } [\![v]\!]\eta = \mathit{int}(0) \\
[\![\texttt{let}\ x \Leftarrow t_1\ \texttt{in}\ t_2]\!]\eta\ \mathsf{h}, &= \bot, \text{ when } [\![t_1]\!]\eta\ \mathsf{h} = \bot \\
[\![\texttt{let}\ x \Leftarrow t_1\ \texttt{in}\ t_2]\!]\eta\ \mathsf{h} &= [\![t_2]\!]\eta[x \mapsto u]\ \mathsf{h}_1 \text{when } [\![t_1]\!]\eta\ \mathsf{h} = (\mathsf{h}_1, u) \\
[\![!v]\!]\eta\ \mathsf{h} &= (\mathsf{h}, \mathsf{h}(\mathsf{l})), \text{ when } [\![v]\!]\eta = \mathit{loc}(\mathsf{l}) \\
[\![v_1 := v_2]\!]\eta\ \mathsf{h} &= (\mathsf{h}[\mathsf{l} \mapsto [\![v_2]\!]\eta], \mathit{int}(0)), \text{ if } [\![v_1]\!]\eta = \mathit{loc}(\mathsf{l}) \\
[\![\texttt{ref}(v)]\!]\eta\ \mathsf{h} &= \mathit{new}(\mathsf{h}, [\![v]\!]\eta) \\
[\![v]\!]\eta &= \mathit{int}(0), \text{ otherwise} \\
[\![t]\!]\eta\ \mathsf{h} &= (\mathsf{h}, \mathit{int}(0)), \text{ otherwise}
\end{aligned}
$$

Fig. 5: Semantics of the untyped meta language

A.2 Setoids

More on dependency We should explain what continuity of a dependent function like $t(-,-)$ is: 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 ascending chains, too, 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 $(x, z, t(p, q))$.

Formally, such dependent functions can be reduced to non-dependent ones using pullbacks, that is t would be a function defined on the pullback of the second and first projections from $\{(x, y, p) \mid p \in A(x, y)\}$ to $|A|$, but we find the dependent notation to be much more readable.

Isomorphic pullbacks

Definition 15. *Let* $\mathbf{W}$ *be a category of worlds. Two pullbacks* $\mathsf{w}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ *and* $\mathsf{w}^{y}_{v}\Diamond^{y'}_{v'}\mathsf{w}'$ *are isomorphic if there is an isomorphism* f *between the two low points of the squares so that* $vf = u$ *and* $v'f = u'$*, thus also* $uf^{-1} = v$ *and* $u'f^{-1} = v'$*.*

It is easy to see that pullback squares can be composed.

Lemma 3. *Given a category of worlds* $\mathbf{W}$*, such that* $\mathsf{w}, \mathsf{w}', \mathsf{w}'' \in \mathbf{W}$*, if* $\mathsf{w}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ *and* $\mathsf{w}'^{y}_{v}\Diamond^{y'}_{v'}\mathsf{w}''$ *are pullback squares as indicated then there exist* z, z', t, t' *such that* $\mathsf{w}^{zx}_{ut}\Diamond^{z'y'}_{v't'}\mathsf{w}''$ *is also a pullback.*

Proof Choose z, z', t, t' in such a way that ${}^{z}_{x'}\Diamond^{z'}_{y}$ and ${}^{u'}_{t}\Diamond^{v}_{t'}$ are pullbacks. The verifications are then an easy diagram chase. □

Pullback squares can be decomposed as formally described below. This property is used for instance in the definition of fibred setoids, formalizing our notion of semantic computation. In particular, to formalize that the executions of related computations do not depend on each other.

Lemma 4. *A pullback square* ${}^{x}_{u}\Diamond^{x'}_{u'}$ *in a category of worlds is isomorphic to* $t({}^{x}_{1}\Diamond^{1}_{x}, {}^{1}_{x'}\Diamond^{x'}_{1})$*.*

Pullback-preserving functors

Lemma 5. *If* A *is a p.p.f.,* $u : \mathsf{w} \to \mathsf{w}'$ *and* $a, a' \in A\mathsf{w}$*, there is a continuous function* $A\mathsf{w}'(u.a, u.a') \to A\mathsf{w}(a, a')$*. Moreover, the "common ancestor"* $\underline{a}$ *of* a *and* a' *is unique up to* $\sim$*.*

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

Definition 16 (Morphism of functors). *If* A, B *are p.p.f., a morphism from* A *to* B *is a pair* $e = (e_0, e_1)$ *of continuous functions where* $e_0 : \Pi\mathsf{w}.A\mathsf{w} \to B\mathsf{w}$ *and* $e_1 : \Pi\mathsf{w}.\Pi\mathsf{w}'.\Pi x : \mathsf{w} \to \mathsf{w}'.\Pi a \in A\mathsf{w}.\Pi a' \in A\mathsf{w}'.A\mathsf{w}'(x.a, a') \to B\mathsf{w}'(x.e_0(a), e_0(a'))$*. A proof that morphisms* e, e' *are equal is given by a continuous function* $\mu : \Pi\mathsf{w}.\Pi a \in A\mathsf{w}.B\mathsf{w}(e(a), e'(a))$*.*

These morphisms compose in the obvious way and so the pullback-preserving functors and morphisms between them form a category.

More on $S(A)$ **and fibred setoids** If ${}^{x}_{u}\Diamond^{x'}_{u'}$ and ${}^{y}_{v}\Diamond^{y'}_{v'}$ are two composable pullback squares with composite ${}^{zx}_{ut}\Diamond^{z'y'}_{v't'}$ and $p \in S(A)\,{}^{x}_{x'}\Diamond^{u}_{u'}(a, a')$ and $p' \in S(A)\,{}^{y}_{y'}\Diamond^{v}_{v'}(a', a'')$, then the

composite proof of $t_{S(A)}(p, p') \in S(A)^{zx}_{ut}\Diamond^{z'y'}_{v't'}(a, a'')$ is given by $t_A(z.p, z'.p')$. Indeed, if $\hat{\mathsf{w}} = \mathrm{cod}(z)$ is the apex of the composite square then $z.p \in A\hat{\mathsf{w}}(zx.a, zx'.a')$ and $z'.p' \in A\hat{\mathsf{w}}(z'y.a', z'y'.a'')$ and $zx'.a' = z'y.a'$ since $zx' = z'y$ so the two proofs compose in $A\hat{\mathsf{w}}$.

Lemma 6. *Let T be a fibred setoid. The elements $\underline{t}$ given by pullback preservation are unique up to $\sim$. If $u : \mathsf{w} \to \mathsf{w}'$ is an isomorphism then there is a continuous function $Tu : T\mathsf{w} \to T\mathsf{w}'$ and it is bijective up to $\sim$ with inverse $T(u^{-1})$. If $\Diamond$ and $\Diamond'$ are isomorphic pullback squares then there are continuous back and forth functions $\Pi t.\Pi t'.T\Diamond(t, t') \to T\Diamond'(t, t')$.*

Lemma 7. *If A is a p.p.f. and T is a fibred setoid then in order to specify a morphism from $S(A)$ to T with given first component $f_0 : \Pi\mathsf{w}.A\mathsf{w} \to T\mathsf{w}$ it is enough to provide a continuous function $f_{0.5} : \Pi\mathsf{w}, \mathsf{w}'.\Pi x : \mathsf{w} \to \mathsf{w}'.\Pi a \in A\mathsf{w}.\Pi a' \in A\mathsf{w}'.A\mathsf{w}'(x.a, a') \to T^{x}_{1}\Diamond^{1}_{x}(f_0(a), f_0(a'))$.*

Proof If (f_0, f_1) is a morphism we can define $f_{0.5}$ by $f_{0.5}(x, p) = f_1(x, a, a', p)$ noting that $p \in S(A)^{x}_{1}\Diamond^{1}_{x}(a, a')$. Conversely, given $f_{0.5}$ to define f_1 we pick a pullback square $\mathsf{w}^{x}_{u}\Diamond^{x'}_{u'}\mathsf{w}'$ with apex $\overline{\mathsf{w}}$ and $a \in A\mathsf{w}, a' \in A\mathsf{w}'$ and $p \in A\overline{\mathsf{w}}(x.a, x'.a')$, i.e., a proof in $S(A)\Diamond(a, a')$. Applying $f_{0.5}$ to $r(-)$ yields the morphism $p_1 \in T^{x}_{1}\Diamond^{1}_{x}(f_0(a), f_0(x.a))$; moreover, applying $f_{0.5}$ to $s(p)$ yields $p_2 \in T^{x'}_{1}\Diamond^{1}_{x'}(f_0(a'), f_0(x.a))$. Then, $t(p_1, s(p_2)) \in Tt(^{x}_{1}\Diamond^{1}_{x}, {}^{1}_{x'}\Diamond^{x'}_{1})(f_0(a), f_0(a'))$ so that Lemmas 4 and 6 yield the desired proof in the square $T^{x}_{u}\Diamond^{x'}_{u'}(f_0(a), f_0(a'))$.

The second part of the lemma about equality is just a restatement of the definition of equality of morphisms of fibred setoids. □

Lemma 8. *Let A, B be p.p.f. For every morphism $e : A \to B$ there is a morphism $S(e) : S(A) \to S(B)$ such that $S(e)_0 = e_0$. Thus, in particular $S(-)$ is a full and faithful functor from the category of p.p.f. on $\mathbf{W}$ to the category of fibred setoids over $\mathbf{W}$.*

On abstract heaps The definition of minimal pullback-preserving functor corresponds to the p.p.f. used for values, but is used for abstract heaps. In particular, an abstract heap at the low-point of a pullback square is the result of forgetting locations from an abstract heap at its apex.

Applying the definition of minimal ppf to the trivial minimal pullback ${}^{u}_{1}\Diamond^{1}_{u}$, plus nonemptiness, yields the following result.

Lemma 9. *For every $u : \mathsf{w} \to \mathsf{w}'$ and $\sigma \in \Sigma\mathsf{w}$ there is morphism of setoids $\Sigma\mathsf{w} \to \Sigma\mathsf{w}'$ which is right inverse to $(-).u$.*

The "unique up to $\sim$" clause allows us in particular to assert the $\sim$-equality of two abstract stores $\sigma, \sigma' \in \Sigma\overline{w}$ by proving $\sigma.x \sim \sigma'.x$ and $\sigma.x' \sim \sigma'.x'$ separately when ${}^{x}_{u}\Diamond^{x'}_{u'}$ is a minimal pullback with apex $\overline{\mathsf{w}}$.

A.3 Computational model

We now discuss a third instantiation of our framework, which captures the setting developed in [5].

Flat stores The *flat stores* instantiation assumes that heap locations contain merely integer values and no pointers. Possible worlds are finite sets of locations together with a function that associates each location a *region* taken from a fixed set *Regs* of regions.

World morphisms must preserve this tagging. We write $\mathsf{l} \in \mathsf{w}$ and $\mathsf{l} \in \mathsf{w}(\mathsf{r})$ to mean that l occurs in w and with region r in the second case. Abstract stores $\mathfrak{S}\mathsf{w}$ comprise those heaps $\mathsf{h} \in \mathbb{H}$ with $\mathrm{dom}(\mathsf{h}) \supseteq \mathsf{w}$ and such that $\mathsf{l} \in \mathsf{w}$ and $\mathsf{h} \in \mathfrak{S}\mathsf{w}$ implies that $\mathsf{h}(\mathsf{l})$ is an integer value, $int(v)$ for $v \in \mathbb{Z}$ (thus all locations hold integer values). We put $\mathsf{h} \sim \mathsf{h}'$ in $\mathfrak{S}\mathsf{w}$ iff for all $\mathsf{l} \in \mathsf{w}$ one has $\mathsf{h}(\mathsf{l}) = \mathsf{h}'(\mathsf{l})$. In this case there is a unique proof, say $\star$. For morphism $u : \mathsf{w} \to \mathsf{w}'$ we define $\mathfrak{S}u : \mathfrak{S}\mathsf{w}' \to \mathfrak{S}\mathsf{w}$ by renaming concrete locations according to u. The elementary effects are $rd_{\mathsf{r}}, wr_{\mathsf{r}}, al_{\mathsf{r}}$ representing reading from within, writing into, allocating within a region r. The associated sets of relations are given by

$$
\begin{aligned}
R \in \mathcal{R}(rd_{\mathsf{r}}) &\iff (\sigma, \sigma') \in R\mathsf{w} \Rightarrow \forall \mathsf{l} \in \mathsf{w}(\mathsf{r}).\sigma(\mathsf{l}) = \sigma'(\mathsf{l}) \\
R \in \mathcal{R}(wr_{\mathsf{r}}) &\iff (\sigma, \sigma') \in R\mathsf{w} \Rightarrow \forall \mathsf{l} \in \mathsf{w}(\mathsf{r}).\forall v \in \mathbb{Z}. \Rightarrow (\sigma[\mathsf{l} \mapsto int(v)], \sigma'[\mathsf{l} \mapsto int(v)]) \in R\mathsf{w} \\
R \in \mathcal{R}(al_{\mathsf{r}}) &\iff (\sigma, \sigma') \in R\mathsf{w} \Rightarrow \forall \mathsf{w}_1.\forall u \in \mathbf{I}(\mathsf{w}, \mathsf{w}_1).(\mathrm{dom}(\mathsf{w}_1) \setminus \mathrm{dom}(\mathsf{w}) \subseteq \mathrm{dom}(\mathsf{w}_1(\mathsf{r}))) \\
&\qquad \Rightarrow \forall \sigma_1 \in \mathfrak{S}\mathsf{w}_1, \sigma_1' \in \mathfrak{S}\mathsf{w}_1'.\sigma_1.u \sim \sigma \wedge \sigma_1'.u \sim \sigma' \wedge \\
&\qquad\quad \forall \mathsf{l} \in \mathrm{dom}(\mathsf{w}_1) \setminus \mathrm{dom}(\mathsf{w}).\sigma_1(\mathsf{l}) = \sigma_1'(\mathsf{l}) \Rightarrow (\sigma_1, \sigma_1') \in R\mathsf{w}_1
\end{aligned}
$$

This essentially mirrors the setting of our earlier relation-based account of reading, writing, and allocation with integer values stores [5] with the difference that allocation is modelled with relations on the same level as reading and writing and that the stores being related share the same layout.

A.4 Proof-relevant logical relations In following establishes that the semantics of the monad corresponds indeed to a semantic computation, that is, a fibred setoid.

Proposition 4. *The semantic computation $T_\varepsilon A$ as defined in Definition 13 is a fibred setoid.*

Proof The tricky case is to show the existence of a transitive operation. It is here that we require the independence of abstract locations as stated in Definition 9, which implies that $\mathfrak{S}$ is also minimal-pullback-preserving.

Assume that there are proofs in $p_1 : T_\varepsilon A {}^{x_1}_{v_1}\!\Diamond^{x_1'}_{v_1'}(\mathsf{c}, \mathsf{c}')$ and $p_2 : T_\varepsilon A {}^{x_2}_{v_2}\!\Diamond^{x_2'}_{v_2'}(\mathsf{c}', \mathsf{c}'')$ where $\mathsf{w} {}^{x_1}_{v_1}\!\Diamond^{x_1'}_{v_1'} \mathsf{w}'$ and $\mathsf{w}' {}^{x_2}_{v_2}\!\Diamond^{x_2'}_{v_2'} \mathsf{w}''$. We also have $\sigma \in \mathfrak{S}\mathsf{w}$ and $\sigma'' \in \mathfrak{S}\mathsf{w}''$, such that they are equivalent in the pullback of the low points of these two pullback squares. Let $\underline{\mathsf{q}}$ be such pullback.

In order to use the proofs p_1 and p_2, we need to construct from σ and σ'' an abstract heap $\sigma' \in \mathfrak{S}\mathsf{w}'$. Let $\overline{\mathsf{q}}$ be the minimal pullback over the apexes of the two pullback squares $\mathsf{w} {}^{x_1}_{v_1}\!\Diamond^{x_1'}_{v_1'} \mathsf{w}'$ and $\mathsf{w}' {}^{x_2}_{v_2}\!\Diamond^{x_2'}_{v_2'} \mathsf{w}''$. Then w and w'' form a pullback square with apex $\overline{\mathsf{q}}$ and low point q. Since $\mathfrak{S}$ is minimal-pullback-preserving, there is a $\sigma_{\mathsf{q}} \in \mathfrak{S}\overline{\mathsf{q}}$, such that it is equivalent to σ and σ'' when taken to the world q. We now define $\sigma' \in \mathfrak{S}\mathsf{w}'$ to be σ_{q} taken to the world w'. We thus have $\sigma' \in \mathfrak{S}\mathsf{w}'$, and $\sigma'' \in \mathfrak{S}\mathsf{w}''$, such that $\sigma.v_1 \sim \sigma'.v_1'$ and $\sigma'.v_2' \sim \sigma''.v_2'$.

We can now use the p_1 and p_2. In particular, let $\mathsf{c}(\sigma) = (\mathsf{w}_1, u_1, \sigma_1, \mathsf{v}_1)$, $\mathsf{c}'(\sigma') = (\mathsf{w}_1', u_1', \sigma_1', \mathsf{v}_1')$, and $\mathsf{c}''(\sigma'') = (\mathsf{w}_1'', u_1'', \sigma_1'', \mathsf{v}_1'')$. From the proofs, we get two pullback squares $\mathsf{w}_1 \Diamond \mathsf{w}_1'$ and $\mathsf{w}_1' \Diamond \mathsf{w}_1''$. It is easy to show that the values obtained are equal in the minimal pullback over the apexes of these two pullback squares and that the abstract heaps are equivalent in the pullback of their low points. □

Definition 17 (cartesian product). *If $(A, \Vdash^A)$ and $(B, \Vdash^B)$ are semantic types their cartesian product $(A \times B, \Vdash^{A\times B})$ is defined by $(A \times B)\mathsf{w} = A\mathsf{w} \times B\mathsf{w}$ (cartesian product of setoids) and $(v_1, v_2) \Vdash^{A\times B}_{\mathsf{w}} (a, b) \iff v_1 \Vdash^A_{\mathsf{w}} a \wedge v_2 \Vdash^B_{\mathsf{w}} b$.*

Definition 18 (function space). *Let* $(A, \Vdash^A)$ *be a semantic type and* $(T, \Vdash^T)$ *be a semantic computation. We define a semantic type* $(A{\Rightarrow}T, \Vdash^{A\Rightarrow T})$ *as follows. An object* f *of* $(A{\Rightarrow}T)\mathsf{w}$ *is a pair* (f_0, f_1) *of continuous functions where* f_0 *assigns to each* w_1 *and* $v : \mathsf{w} \to \mathsf{w}_1$ *a continuous function* $f_0(v) : A\mathsf{w}_1 \to T\mathsf{w}_1$*. The second component* f_1 *assigns to each* $v : \mathsf{w} \to \mathsf{w}_1$ *and* $v_1 : \mathsf{w}_1 \to \mathsf{w}_2$ *a continuous function* $\Pi a \in A\mathsf{w}_1.\Pi a' \in A\mathsf{w}_2.A\mathsf{w}_2(v_1.a, a') \to T\,{}^{v_1}_{1}\!\Diamond^{1}_{v_1}(f_0(v, a), f_0(v_1 v, a'))$*.*

If $f, f' \in |A{\Rightarrow}T|$ *then a proof* $\mu \in (A{\Rightarrow}T)(f, f')$ *is a continuous function assigning to each* $v : \mathsf{w} \to \mathsf{w}_1$ *and* $a \in A\mathsf{w}_1$ *a proof* $\mu(v, a) \in T\,{}^{1}_{1}\!\Diamond^{1}_{1}(f_0(v, a), f'_0(v, a))$*.*

If $u : \mathsf{w} \to \mathsf{w}'$ *and* $f = (f_0, f_1) \in (A{\Rightarrow}T)\mathsf{w}$ *then* $u.f \in (A{\Rightarrow}T)\mathsf{w}'$ *is given by precomposition with* u*, i.e.,* $(u.f)_0(v, a) = f_0(vu, a)$*, etc.*

As for the realisation relation $\Vdash^{A\Rightarrow T}$ *we put* $v \Vdash^{A\Rightarrow T}_{\mathsf{w}} f$ *to mean that* $v = \mathit{fun}(g)$ *for some* g *and whenever* $i : \mathsf{w} \to \mathsf{w}_1$ *is an inclusion and* $u \Vdash^{A}_{\mathsf{w}_1} a$ *then* $g(u) \Vdash^{T}_{\mathsf{w}_1} f(i, a)$*.*

Notice that unlike morphisms the elements of the function space are *not* identified if they are "provably equal." Notice also that if $v \Vdash^{A\Rightarrow T}_{\mathsf{w}} f$ implies $v \Vdash^{A\Rightarrow T}_{\mathsf{w}_1} i.f$ whenever $i : \mathsf{w} \to \mathsf{w}_1$ is an inclusion.

In what follows we define semantic counterparts to the generic syntactic constructions common to all instantiations, namely application and abstraction, sequential composition, subeffecting, and recursion that allow us to define this interpretation of derivations in a compositional fashion. Having given these semantic counterparts we then omit the formal definition of the interpretation $[\![-]\!]$.

Lemma 10 (Abstraction). *Let* Γ, A *be semantic types,* T *a semantic computation. There is a function* λ *so that if* $e : S(\Gamma \times A) \to T$ *is a morphism of fibred setoids then* $\lambda(e) : S(\Gamma) \to A{\Rightarrow}T$*. Moreover, if* $e \sim e'$ *then* $\lambda(e) \sim \lambda(e')$ *and if* $f \Vdash^{\Gamma\times A\to T} e$ *then* $\lambda\eta.\lambda a.f(\eta, a) \Vdash^{\Gamma\to A\Rightarrow T} \lambda(e)$*.*

Lemma 11 (Application). *Let* A *be a semantic type and* T *be a semantic computation. There is a morphism* $\mathit{app} : S((A{\Rightarrow}T) \times A) \to T$ *and* $\lambda(f, a).f(a) \Vdash^{((A\Rightarrow T)\times A)\to T} \mathit{app}$*.*

Lemma 12 (subeffecting). *Let* Γ, A *be semantic types and* $\varepsilon, \varepsilon'$ *be effects. There is a function* subeff*, so that if* $e : S(\Gamma) \to T_\varepsilon A$*, then* $\mathrm{subeff}(e) : S(\Gamma) \to T_{\varepsilon\cup\varepsilon'}A$*. Moreover, if* $e \sim e'$*, then* $\mathrm{subeff}(e) \sim \mathrm{subeff}(e')$*. Finally, if* $f \Vdash^{\Gamma\to T_\varepsilon A} e$ *then* $f \Vdash^{\Gamma\to T_{\varepsilon\cup\varepsilon'}A} \mathrm{subeff}(e)$*.*

Proof For the first component, subeff_0, we use the same first component e_0 of e. What changes is the definition of the second component, subeff_1. It is defined only for relations $R \in \mathcal{R}(\varepsilon \cup \varepsilon')$, for which e_1 is also defined. For some related given abstract heaps in R, subeff_1 calls e_1 constructing the corresponding pullback. For proofs the reasoning is similar. □

We elide assertions about $\sim$-versions of beta-eta-equality, and the existence of "value morphisms" of type $S(A) \to T_\varepsilon A$ for any semantic type A.

Lemma 13 (let). *Let* Γ, A, B *be semantic types and* ε *an effect. There is a function let such that if* $e_1 : S(\Gamma) \to T_\varepsilon A$ *and* $e_2 : S(\Gamma \times A) \to T_\varepsilon B$ *are morphisms then* $\mathit{let}(e_1, e_2) : S(\Gamma) \to T_\varepsilon B$*. Moreover, if* $e_1 \sim e'_1$ *and* $e_2 \sim e'_2$ *then* $\mathit{let}(e_1, e_2) \sim \mathit{let}(e'_1, e'_2)$*. Finally, if* $f_1 \Vdash^{\Gamma\to T_\varepsilon A} e_1$ *and* $f_2 \Vdash^{\Gamma\times A\to T_\varepsilon B} e_2$ *then* $\lambda\eta.\lambda\mathsf{h}.\mathit{let}\ (\mathsf{h}_1, v){=}f_1(\eta)(\mathsf{h})\ \mathit{in}\ f_2(\eta, v)(\mathsf{h}_1) \Vdash^{\Gamma\to T_\varepsilon A} \mathit{let}(e_1, e_2)$*.*

Proof Consider the following definition for the first component of the morphism $let(e_1, e_2)$ which is only defined when e_1 and e_2 are defined. The type of this component is $[\![\Gamma]\!]\mathsf{w} \to T_\varepsilon[\![B]\!]\mathsf{w}$. Hence, assume a world w, and a context $\gamma \in [\![\Gamma]\!]\mathsf{w}$, then one returns an object $(\mathsf{c}_0, \mathsf{c}_1) \in T_\varepsilon[\![B]\!]\mathsf{w}$. The first component c_0 is: $\Pi\mathsf{w}.\Pi\gamma \in [\![\Gamma]\!]\mathsf{w}.\Pi\sigma \in \Im\mathsf{w}.e_2(\mathsf{w}_1)(\gamma, \mathsf{v}_1)\sigma_1$ where $e_1(\mathsf{w})(\gamma)\sigma = (\mathsf{w}_1, u_1, \sigma_1, \mathsf{v}_1)$.

For the second component, c_1, assume a relation $R \in \mathcal{R}(\varepsilon)$, and two abstract heaps $\sigma, \sigma' \in \Im\mathsf{w}$ such that $(\sigma, \sigma') \in R\mathsf{w}$. From e_1 we get a proof $\mathsf{w}_1 \,{}^{x_1}_{v_1}\!\Diamond^{x'_1}_{v'_1}\, \mathsf{w}'_1$, where $e_1(\mathsf{w})(\gamma)\sigma = (\mathsf{w}_1, u_1, \sigma_1, \mathsf{v}_1)$ and $e_1(\mathsf{w})(\gamma)\sigma' = (\mathsf{w}'_1, u'_1, \sigma'_1, \mathsf{v}'_1)$, such that $(\sigma_1.v_1, \sigma'_1.v'_1) \in R$ and $p : [\![A]\!]\overline{\mathsf{w}_1}(x_1.\mathsf{v}_1, x'_1.\mathsf{v}'_1)$. Applying e_2 on $\sigma_1.v_1$ and $\sigma'_1.v'_1$ we get a proof $\mathsf{q}_2 \,{}^{y_2}_{v_2}\!\Diamond^{y'_2}_{v'_2}\, \mathsf{q}'_2$, such that $(\tilde{\sigma_2}.v_2, \tilde{\sigma_2}'.v'_2) \in R$. However, we need to show that the heaps obtained from applying e_2 on σ_1 and σ'_1 (using the correct world and context), namely σ_2 and σ'_2, are related. For this we rely on the morphism $(e_2)_1$. In particular, we use $(e_2)_1$ on the pullback $\mathsf{w}_1 \,{}^{1}_{x_1}\!\Diamond^{x_1}_{1}\, \underline{\mathsf{w}_1}$ and obtain a pullback $\mathsf{w}_2 \Diamond \mathsf{q}_2$ such that σ_2 and $\tilde{\sigma_2}$ are equal in its low point. Similarly, applying $(e_2)_1$ on the pullback $\underline{\mathsf{w}_1} \,{}^{x'_1}_{1}\!\Diamond^{1}_{x'_1}\, \mathsf{w}'_1$, we get a pullback $\mathsf{q}'_2 \Diamond \mathsf{w}'_2$, where $\tilde{\sigma_2}'$ is equal to σ'_2 in its pullback. Using Lemma 3, we compose the pullbacks $\mathsf{w}_2 \Diamond \mathsf{q}_2$, $\mathsf{q}_2 \Diamond \mathsf{q}'_2$ and $\mathsf{q}'_2 \Diamond \mathsf{w}'_2$, obtaining a common pullback $\underline{\mathsf{q}}$, where σ_2 and σ'_2 when taken to $\underline{\mathsf{q}}$ are in R.

The morphism $let(e_1, e_2) \sim let(e'_1, e'_2)$ can be then defined when $e_1 \sim e'_1$ and $e_2 \sim e'_2$ are defined. Assume a pullback $\mathsf{w}_1 \,{}^{1}_{1}\!\Diamond^{1}_{1}\, \mathsf{w}$ and an abstract heap $\sigma \in \Im\mathsf{w}$ and a context $\gamma \in [\![\Gamma]\!]\mathsf{w}$. Using the morphism between e_1 and e'_1 on these objects, we obtain a pullback $\mathsf{w}_1 \,{}^{x_1}_{v_1}\!\Diamond^{x'_1}_{v'_1}\, \mathsf{w}'_1$, $p_1 \in [\![A]\!]\overline{\mathsf{w}_1}(x_1.\mathsf{v}_1, x'_1.\mathsf{v}'_1)$ and $q_1 : \sigma_1.v_1 \sim \sigma'_1.v'_1$, where $e_1(\mathsf{w})(\gamma)\sigma = (\mathsf{w}_1, u_1, \sigma_1, \mathsf{v}_1)$ and $e'_1(\mathsf{w})(\gamma)\sigma = (\mathsf{w}'_1, u'_1, \sigma'_1, \mathsf{v}'_1)$. From the pullback preserving property of computations and p_1, there is a common value $\underline{\mathsf{v}} \in [\![A]\!]\underline{\mathsf{w}_1}$ and context $\underline{\gamma} \in [\![\Gamma]\!]\underline{\mathsf{w}_1}$ which are equal, respectively, to v_1 and v'_1, and γ and γ' (when taken to the correct world). We then construct a proof $[\![\Gamma \times A]\!]\underline{\mathsf{w}_1}$. We now apply twice the morphism between e_2 and e'_2 once in the pullback $\mathsf{w}_1 \Diamond \underline{\mathsf{w}_1}$ and another on the pullback $\underline{\mathsf{w}_1} \Diamond \mathsf{w}'_1$, obtaining two pullbacks $\mathsf{w}_2 \Diamond \mathsf{q}_2$ and $\mathsf{q}_2 \Diamond \mathsf{w}'_2$. From Lemma 3, we can compose them where the resulting values and heaps are equal. □

Lemma 14 (fix). *Let Γ, D be semantic types so that for each w the predomain $D\mathsf{w}$ is a domain with least element $\bot\mathsf{w}$ such that $(\bot\mathsf{w}, \bot\mathsf{w}, r(\bot\mathsf{w})) \leq (d, d', p)$ holds for every proof $p \in D(d, d')$ and such that $x.\bot_{\mathsf{w}} = \bot_{\mathsf{w}'}$ holds for every $x : \mathsf{w} \to \mathsf{w}'$.*[1]

i There then exists a function fix so that whenever $e : \Gamma \times D \to D$ then $fix(e) : \Gamma \to D$

ii If $e \sim e'$ then $fix(e) \sim fix(e')$. Furthermore, the fixpoint and unrolling equations from Lemma 14 hold.

iii Finally, if $f \Vdash^{\Gamma \times D \to D} e$ then $f^\dagger \Vdash fix(e)$.

Proof For every w we have $e_0\mathsf{w} : \Gamma\mathsf{w} \times D\mathsf{w} \to D\mathsf{w}$. We can thus form $fix(e)_0\mathsf{w} := (e_0\mathsf{w})^\dagger : \Gamma\mathsf{w} \to D\mathsf{w}$. It remains to define $fix(e)_1$. To do that, we recall that we have an ascending chain of elements $fix^n(e)_0\mathsf{w}(\gamma) \in D\mathsf{w}$ given by $fix^0(e)_0\mathsf{w}(\gamma) = \bot_{\mathsf{w}}$ and $fix^{n+1}(e)_0\mathsf{w}(\gamma) = e_0\mathsf{w}(\gamma, fix^n(e)_0\mathsf{w}(\gamma))$ and have $fix(e)_0\mathsf{w}(\gamma) = \sup_n fix^n(e)_0\mathsf{w}\gamma$. Now suppose that $\gamma \in \Gamma\mathsf{w}$ and $x : \mathsf{w} \to \mathsf{w}'$ and $\gamma' \in \Gamma\mathsf{w}'$ and $p \in \Gamma\mathsf{w}'(x.\gamma, \gamma')$. Write $d_n = fix^n_0\mathsf{w}(\gamma)$ and $d'_n = fix^n_0\mathsf{w}'(\gamma')$. Inductively, we get proofs $p_n \in D\mathsf{w}'(x.d_n, d'_n)$ where $p_0 = r(\bot_{\mathsf{w}'})$ (note that $x.\bot_{\mathsf{w}} = \bot_{\mathsf{w}'}$) and $p_{n+1} = e_1(p, p_n)$. Since $(x.\bot_{\mathsf{w}}, \bot_{\mathsf{w}'}, r(\bot_{\mathsf{w}'})) \leq$

[1] For example $D = A \Rightarrow T_\varepsilon B$ for semantic types A, B.

$(x.d_1, d'_1, p_1)$ we obtain by monotonicity of e_1 and induction that $(x.d_n, d'_n, p_n)$ is an ascending chain with supremum $(x.\sup_n d_n, \sup_n d'_n, q)$ for some proof q which we take as $\mathit{fix}(e)_1(p)$. Note that the passage from p to q is continuous. □

A.5 Applications

The following lemma formalizes our intuition that

Lemma 2 Proof The proof that the values are equal in $\underline{\mathsf{w}}$ follows directly from the definition of computations and effects.

For the first part, we use the following relation R defined for all worlds w_1, such that $u : \mathsf{w} \to \mathsf{w}_1$:

$\{(\sigma, \sigma') \mid \sigma \sim_{\mathrm{rds}(\varepsilon,\mathsf{w})} \sigma' \wedge \forall \mathsf{l} \in \mathsf{w}.$
$(\sigma.u, \sigma_0) \in \mathsf{l}^R \wedge (\sigma'.u, \sigma'_0) \in \mathsf{l}^R \vee (\sigma.u, \sigma'.u) \in \mathsf{l}^R\}$

Otherwise, for the worlds w_2 not reachable from w, the relation $R\mathsf{w}_2$ is the trivial set. Notice that $R \in \mathcal{R}(\varepsilon)$ and it is contravariant. The claim then follows directly.

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

$\{(\sigma, \sigma') \mid \sigma \sim_{\mathrm{rds}(\varepsilon,\mathsf{w})} \sigma' \wedge \sigma \sim_{\mathit{nwrs}(\varepsilon,\mathsf{w})} \sigma_0.u\}$

And we use a similar relation for showing that σ'_0 and $\sigma'_1.u'$ agree on the not written locations $\mathit{nwrs}(\varepsilon, \mathsf{w})$.

For the third property, first, we show that there is an isomorphism between $\mathsf{w}(\mathsf{r})$ and $\underline{\mathsf{w}}(\mathsf{r})$ for all regions $\mathsf{r} \notin \mathrm{als}(\mathsf{r})$ by using the following relation:

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

where $\#_{\mathsf{r}}$ denotes the number of abstract locations coloured with r. Clearly, $R \in \mathcal{R}(\varepsilon)$ as ε does not contain any allocation effects. This gives us one direction, while the other direction is obtained by using the inclusion morphisms. Given this property, one can easily construct the function c'. □

Proposition 5. *(commuting computations) Suppose that:* $\Gamma \vdash e_1 : \tau_1 \;\&\; \varepsilon_1$ *and* $\Gamma \vdash e_2 : \tau_2 \;\&\; \varepsilon_2$, *where* $\mathrm{rds}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \mathrm{rds}(\varepsilon_2) \cap \mathrm{wrs}(\varepsilon_1) = \mathrm{wrs}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset$. *Let*

$$e = \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow e_2\ \mathtt{in}\ (x, y) \quad \textit{and} \quad e' = \mathtt{let}\ y \Leftarrow e_2\ \mathtt{in}\ \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ (x, y)$$

then $[\![\Gamma \vdash e : \tau_1 \times \tau_2 \;\&\; \varepsilon_1 \cup \varepsilon_2]\!] \sim [\![\Gamma \vdash e' : \tau_1 \times \tau_2 \;\&\; \varepsilon_1 \cup \varepsilon_2]\!]$.

Proof Assume a world w and a context $\gamma \in [\![\Gamma]\!]\mathsf{w}$. Let $\mathsf{c}_i = [\![\Gamma \vdash e_i : \tau_i \;\&\; \varepsilon_i]\!]$ for $i = 1, 2$.

It is enough to assume a pullback $\mathsf{w}_1^1 \Diamond_1^1 \mathsf{w}$, and an abstract heap $\sigma_0 \in \Im\mathsf{w}$. Assume that these functions are defined as follows:

$\mathsf{c}_1(\mathsf{w})(\gamma)\sigma_0 = (\mathsf{w} \uplus \mathsf{w}_1, u_1, \sigma_1, \mathsf{v}_1)$
$\mathsf{c}_2(\mathsf{w} \uplus \mathsf{w}_1)(u_1.\gamma)\sigma_1 = (\mathsf{w} \uplus \mathsf{w}_1 \uplus \mathsf{w}_2, u_2, \sigma_2, \mathsf{v}_2)$
$\mathsf{c}'_2(\mathsf{w})(\gamma)\sigma_0 = (\mathsf{w} \uplus \mathsf{w}'_1, u'_1, \sigma'_1, \mathsf{v}'_1)$
$\mathsf{c}'_1(\mathsf{w} \uplus \mathsf{w}'_2)(u'_1.\gamma)\sigma'_1 = (\mathsf{w} \uplus \mathsf{w}'_1 \uplus \mathsf{w}'_2, u'_2, \sigma'_2, \mathsf{v}'_2)$

One can easily show that when one of these functions is undefined, then the corresponding function is also undefined.

We need to show that there is a proof $\mathsf{w} \uplus \mathsf{w}_1 \uplus \mathsf{w}_2 {}^{x}_{v}\Diamond^{x'}_{v'} \mathsf{w} \uplus \mathsf{w}'_1 \uplus \mathsf{w}'_2$ such that $p : \sigma_2.v \sim \sigma'_2.v'$ and $p_1 : xu_2.\mathsf{v}_1 \sim x'.\mathsf{v}'_2$ and $p_2 : x.\mathsf{v}_2 \sim x'u'_2.\mathsf{v}'_1$. Decompose $\mathsf{w} = \mathsf{w}_0 \uplus \mathsf{q}_1 \uplus \mathsf{q}_2$, where $\mathsf{w}(\mathrm{wrs}(\varepsilon_i)) \subseteq \mathsf{q}_i$. The existence of such decomposition follows from the disjointness of write effects in ε_1 and ε_2.

From Lemma 2 and from the disjointness of reads and writes, it is the case that σ_0 and σ'_1 agree on the locations in $\mathsf{w}_0 \uplus \mathsf{q}_1$. That is, there is a proof $p : \sigma_0.1 \sim \sigma'_1.x_1$,

defined using the proof $\mathsf{w}_0 \uplus \mathsf{q}_1 \,{}^{x_1}_{1}\!\Diamond^{1}_{x_1}\, \mathsf{w}_0 \uplus \mathsf{q}_1 \uplus \mathsf{w}'_2$, where $x_1 : \mathsf{w}_0 \uplus \mathsf{q}_1 \to \mathsf{w}_0 \uplus \mathsf{q}_1 \uplus \mathsf{w}'_2$. Applying $(e_1)_1$ to the objects above, we get the pullback $\mathsf{w}_0 \uplus \mathsf{q}_1 \uplus \mathsf{w}_1 \,{}^{x_2}_{v_2}\!\Diamond^{x'_2}_{v'_2}\, \mathsf{w}_0 \uplus \mathsf{q}_1 \uplus \mathsf{w}'_2 \uplus \mathsf{w}'_1$, and proof $q : x_2.\mathsf{v}_1 \sim x'_2.\mathsf{v}'_2$. Symmetrically, we obtain the proofs $\mathsf{w}_0 \uplus \mathsf{q}_2 \uplus \mathsf{w}_2 \,{}^{x_3}_{v_3}\!\Diamond^{x'_3}_{v'_3}\, \mathsf{w}_0 \uplus \mathsf{q}_2 \uplus \mathsf{w}'_1 \uplus \mathsf{w}'_2$, and $q' : x_3.\mathsf{v}_2 \sim x'_3.\mathsf{v}'_1$. Hence, there is also a proof in the larger world $cod(x)$.

To see informally that the final heaps σ_2 and σ'_2 are equal, we use the following facts obtained using Lemma 2: σ_2 and σ_1 agree on the locations in $\mathsf{w}_0 \uplus \mathsf{q}_1$; moreover, σ'_2 and σ_1 agree on the locations in $\mathsf{w}_0 \uplus \mathsf{q}_1$; hence σ_2 and σ'_2 agree on the locations in $\mathsf{w}_0 \uplus \mathsf{q}_1$. Symmetrically, we can also argue that σ_2 and σ'_2 agree on the locations in $\mathsf{w}_0 \uplus \mathsf{q}_2$. Composing these proofs (see comment after Lemma 9 why this is allowed), we get that σ_2 and σ'_2 agree on the locations in w. Finally, since the locations allocated by one computation are not used by the other computation, the final heaps are equal at the apex world. □

The following propositions are also provable. All propositions are proved in a similar way as the soundness proof of the commuting case, using Lemma 2 when needed. For instance, the soundness proof of the duplicated computation uses the third case in Lemma 2.

Proposition 6 (dead computation). *Suppose that* $\Gamma \vdash e : \mathtt{unit}\ \&\ \varepsilon$*, that* $\mathrm{wrs}(\varepsilon) = \emptyset$ *and that* $[\![\Gamma \vdash e : \mathtt{unit}\ \&\ \varepsilon]\!]\mathsf{w}(\gamma)(\sigma)$ *is defined for all* $\mathsf{w}, \gamma \in [\![\Gamma]\!]\mathsf{w}, \sigma \in \Subset\mathsf{w}$*. Then if for all worlds* w*, all contexts* $\gamma \in [\![\Gamma]\!]\mathsf{w}$*, and abstract heaps* $\sigma \in \Subset\mathsf{w}$*, the function* $[\![\Gamma \vdash e]\!](\mathsf{w})(\gamma)(\sigma)$ *is defined, then* $[\![\Gamma \vdash e : \mathtt{unit}\ \&\ \varepsilon]\!] \sim [\![\Gamma \vdash () : \mathtt{unit}\ \&\ \varepsilon]\!]$.

Proof Assume a world w and a context $\gamma \in [\![\Gamma]\!]\mathsf{w}$. Let $\mathsf{c} = [\![\Gamma \vdash e : \tau\ \&\ \varepsilon]\!]$. It is enough to assume a pullback $\mathsf{w}_1^1\Diamond_1^1\mathsf{w}$, and an abstract heap $\sigma_0 \in \Subset\mathsf{w}$. Let $\mathsf{c}(\mathsf{w})(\gamma)\sigma_0 = (\mathsf{w}, 1, \sigma_1, \mathsf{v}_1)$. We need to construct a pullback such that v_1 is equivalent to () in its apex and σ_1 is equivalent to σ_0 in its low point. Consider the pullback $\mathsf{w}_1 \,{}^{1}_{u}\!\Diamond^{u}_{1}\, \mathsf{w}$. Clearly $\mathsf{v}_1 = ()$, and therefore the values are equivalent in w_1. Moreover, from the fact that $\mathrm{wrs}(\varepsilon) = \emptyset$, σ_1 and σ_0 agree on all locations in w. Hence, $\sigma_1.u \sim \sigma_0$, which finishes the proof. □

Proposition 7 (duplicated computation). *Suppose that* $\Gamma \vdash e : \tau\ \&\ \varepsilon$ *and suppose that* $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \mathrm{als}(\varepsilon) = \emptyset$*. Thus, e reads and writes on disjoint portions of the store and makes no allocations. The the terms* e_1 *and* e_2 *below*

$$\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ (x, x) \quad \textit{and} \quad \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ (x, y)$$

are contextually equivalent. That is formally $[\![\Gamma \vdash e_1 : \tau \times \tau\ \&\ \varepsilon]\!] \sim [\![\Gamma \vdash e_2 : \tau \times \tau\ \&\ \varepsilon]\!]$.

Proof Assume a world w and a context $\gamma \in [\![\Gamma]\!]\mathsf{w}$. Let $\mathsf{c} = [\![\Gamma \vdash e : \tau\ \&\ \varepsilon]\!]$. It is enough to assume a pullback $\mathsf{w}_1^1\Diamond_1^1\mathsf{w}$, and an abstract heap $\sigma_0 \in \Subset\mathsf{w}$. From Lemma 2 and since these functions do not allocate, we can assume that they do not cause any world extension and are therefore defined as follows:

$$\mathsf{c}(\mathsf{w})(\gamma)\sigma_0 = (\mathsf{w}, 1, \sigma_1, \mathsf{v}_1) \text{ and } \mathsf{c}(\mathsf{w})(\gamma)\sigma_1 = (\mathsf{w}, 1, \sigma_2, \mathsf{v}_2).$$

We need to show that the values v_1 and v_2 are equivalent and the heaps σ_1, obtained by applying once e, and σ_2, obtained by applying twice e, are also equal.

Decompose $\mathsf{w} = \mathsf{w}_0 \uplus \mathsf{w}_r \uplus \mathsf{w}_w$, where w_r contains all the regions read by e and w_w all the regions written by e. This is possible because of the disjointness of of e's read and write effects. From Lemma 2 and the disjointness of e's read and write effects, we have that σ_0 and σ_1 agree on the regions read by e, that is, $\sigma_0 \sim_{\mathrm{rds}(\varepsilon,\mathsf{w})} \sigma_1$. Hence, again from Lemma 2, we have that the values v_1 and v_2 are equal. Moreover, the locations in w_w are equaly written, while the locations in $\mathsf{w}_0 \uplus \mathsf{w}_r$ are left unchanged, that is, σ_1 and σ_2 agree on the location in w. □

Proposition 8 (pure lambda hoist). *Suppose that $\Gamma \vdash e : Z \,\&\, \emptyset$ and $\Gamma, x{:}X, y{:}Z \vdash e' : Y \,\&\, \varepsilon$ Let e_1 and e_2 be respectively $\lambda x.\mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ e'$ and $\mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ \lambda x.e'$. Then $[\![\Gamma \vdash e_1 : (X \xrightarrow{\varepsilon} Y) \,\&\, \emptyset]\!] \sim [\![\Gamma \vdash e_2 : (X \xrightarrow{\varepsilon} Y) \,\&\, \emptyset]\!]$.*

Proof Assume a world w and a context $\gamma \in [\![\Gamma]\!]\mathsf{w}$. Let $\mathsf{c} = [\![\Gamma \vdash e : \tau \,\&\, \varepsilon]\!]$ and $\mathsf{c}' = [\![\Gamma, x : X, y : Z \vdash e' : \tau \,\&\, \varepsilon]\!]$. It is enough to assume a pullback $\mathsf{w}_1^1 \diamond_1^1 \mathsf{w}$, and an abstract heap $\sigma_0 \in \Im\mathsf{w}$. Since e has no effects, we have no world extension:

$$\mathsf{c}(\mathsf{w})(\gamma)\sigma_0 = (\mathsf{w}, 1, \sigma_1', \mathsf{v}_1')$$

Moreover, from Lemma 2, σ_1 and σ_0 agree on all locations. We now show that

$$[\![\Gamma \vdash \lambda x.\mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ e'(x,y) : (X \xrightarrow{\varepsilon} Y)]\!] \sim [\![\Gamma \vdash \lambda x.e'(x, \mathsf{v}_1') : (X \xrightarrow{\varepsilon} Y)]\!]$$

In order to prove this, assume a morphism $v : \mathsf{w} \to \mathsf{w}_1$ and $a \in [\![X]\!]\mathsf{w}_1$. We need then to prove that the computations resulting from applying a to the functions above are equivalent in the pullback $\mathsf{w}_1 {}_1^1\diamond_1^1 \mathsf{w}_1$. For this, assume an abstract heap $\sigma \in \Im\mathsf{w}_1$. Since e has no effect, we have no world extension:

$$\begin{aligned}
&\mathsf{c}(\mathsf{w}_1)(\gamma)\sigma = (\mathsf{w}_1, 1, \sigma_1, \mathsf{v}_1)\\
&\mathsf{c}'(\mathsf{w}_1)(\gamma, a, \mathsf{v}_1)\sigma_1 = (\mathsf{w}_2, 1, \sigma_2, \mathsf{v}_2)\\
&\mathsf{c}'(\mathsf{w}_1)(\gamma, a, \mathsf{v}_1')\sigma = (\mathsf{w}_2', 1, \sigma_1', \mathsf{v}_2)
\end{aligned}$$

Since e is pure, we have $\mathsf{v}_1 = v.\mathsf{v}_1'$ and from Lemma 2 we have that σ_1 and σ agree on all locations in w_1 and in particular on locations read by e'. Hence, again by Lemma 2 the pullback proof exists where σ_2 and σ_1' are equal in its low point and the resulting values are equal in its apex. □

Masking We now justify soundness of the masking rule shown below:

$$\frac{\Gamma \vdash t : \tau \,\&\, \varepsilon \quad \mathsf{r} \notin \mathrm{regs}(\Gamma) \cup \mathrm{regs}(\tau)}{\Gamma \vdash t : \tau \,\&\, \varepsilon \setminus \{rd_\mathsf{r}, wr_\mathsf{r}, al_\mathsf{r}\}}\ \text{Masking}$$

which allows one to mask effects, that is, allowing it to behave closer to pure functions. As discussed in [4], as the effect-dependent equations can be applied only if some conditions on the set of effects is satisfied, the masking of effects may enable the use of such equations. (See the commutation computation equation.)

Assume that for for every set of regions R, we take a different instantiation $\mathbf{W}_R$ where all abstract locations get colors from R. Within $\mathbf{W}_R$ we can interpret app, lambda, fix, etc. If $R \subseteq R'$ and X is a semantic type over $\mathbf{W}_{R'}$ denote $X|R$ its restriction to $\mathbf{W}_R$. In our setting, we prove of the soundness of the masking rule by providing morphisms between the objects in $\mathbf{W}_R$ and objects in $\mathbf{W}_{R'}$ when restricted to R, where $R \subseteq R'$.

Body of Loop	Prolog	Steady Program	Epilogue
`x := load(p);`	`p1 := p;`	`store(p1, y);` $[wr_{r_1}]$	`store(p1, y);` $[wr_{r_1}]$
`y := x * c;`	`p2 := p;`	`p1 := p2 + 8;`	`y := x2 * c;`
`store(p, y);`	`x1 := x;`	`y := x2 * c;`	`store(p2, y);` $[wr_{r_2}]$
`p := p + 8;`	`x2 := x;`	`x1 := load(p1);` $[rd_{r_1}]$	`x := x2;`
`i := i + 1;`	`x1 := load(p1);` $[rd_{r_1}]$	`store(p2, y);` $[wr_{r_2}]$	`p := p2;`
	`p2 := p1 + 8;`	`p2 = p1 + 8;`	
	`x2 := load(p2);` $[rd_{r_2}]$	`y = x1 * c;`	
	`y := x1 * c;`	`y = load(p2);` $[rd_{r_2}]$	
	`i := i + 2;`	`i := i + 2;`	

Fig. 6: Program obtained from the loop unrolling technique. Here `p`, `p1` and `p2` are pointers and all `load` and `store` operations are on 64 bit numbers (float).

This corresponds in our setting to the Masking Lemma in [4] and is formalized by introducing the notion of matching pairs: Let X be a semantic type over $\mathbf{W}_R$ and X' be a semantic type over $\mathbf{W}_{R'}$. The two form a *matching pair* if there are morphisms $i : X \to X'|R$ and $j : X'|R \to X$ both tracked by the identity on the level of values and isomorphisms w.r.t. $\sim$. The idea is that if τ only mentions regions in R then $[\![\tau]\!]$ with respect to R and $[\![\tau]\!]$ with respect to R' will be a matching pair.

Suppose that $\mathsf{w} \in \mathbf{W}_R$. If $\sigma \in \Im\mathsf{w}$ then, since w can be viewed also over R', we can understand σ as living in $\mathbf{W}_{R'}$. Conversely, if $\mathsf{w} \in \mathbf{W}_{R'}$ and $\sigma \in \Im\mathsf{w}$, then we also have $\sigma \in \Im\mathsf{w}|R$ by coarsening. This is because if σ satisfies all the contracts in the larger worlds involving the regions R', then it also satisifies the contracts for the regions in the smaller set R. In fact, every world $\mathsf{w} \in \mathbf{W}_{R'}$ induces a world $\mathsf{w}|R \in \mathbf{W}_R$.

We now prove that if only regions from R are mentioned in τ then $[\![\tau]\!]R$ and $[\![\tau]\!]R'$ form a matching pair where $[\![\cdot]\!]R$ denotes the interpretation with respect to $\mathbf{W}_R$: Suppose that ε mentions all of R' and that $(\Gamma, \Gamma'), (A, A')$ are matching pairs and that $e : \Gamma' \to T_\varepsilon A'$ is a morphism tracked by $f : \mathbb{V} \to \mathbb{C}$. There then exists a morphism $mask(e) : \Gamma \to T_{\varepsilon|R} A$ also tracked by f and if $e \sim e'$ then $mask(e) \sim mask(e')$.

Let the morphisms i_Γ and j_Γ due to the fact that (Γ, Γ') form a matching pair and i_A and j_A due to the fact that (A, A') form a matching pair. It is then easy to prove the soundness of masking by using the morphism $mask(e)\mathsf{w}(\gamma)(\sigma) = \texttt{let}\ (\sigma_1, v) \Leftarrow e(i_\Gamma(\gamma))(\sigma)\ \texttt{in}\ (\sigma_1, j_A(v))$.

Example: Loop Unrolling Loop unrolling is a software pipelining technique used to enhance the use of parallel processing. The idea is instead of iterating a loop in a sequential manner, one attempts to process a number of iterations of the loop at the same time using multiple processors.

As described in [31] implementing and proving the correctness of loop unrolling techniques is hard as one needs to demonstrate that the program resulting from loop unrolling that can be executed in parallel is equivalent to the original sequential program. We briefly illustrate the power of our system with regions and effects by one of the running examples in [31]. Consider a loop program whose body is depicted in Figure 6. Intuitively, this program is multiplying all the elements of an array of float values by the value `c`. Clearly, instead of executing this program sequentially, we can execute different iterations in parallel. In particular, after applying the loop unrolling optimization to a program, one obtains a program that is divided in three parts: the prolog, that initializes all the variables, the steady state, that is iterated, and the epilogue, that is executed when the loop condition is no longer true and the loop is over. Figure 6

contains the program obtained by loop unrolling two iterations of the program above. The Prolog and the Epilogue are executed at the beginning and the end, respectively, while the Steady Program may be executed several times.

The task is to show that the optimized program is equivalent to the sequential program above. Using the unrolling equations from Lemmas 14 we can unroll the loop twice ($n = 2$) and extract a prologue. We can then conclude with effect-dependent equivalences, in particular Prop. 5 as follows. We use two regions r_1 and r_2. All even elements of the array, that is, `p`, `p + 16`, `p + 32`, ..., belong to the region r_1, while all odd elements, that is, `p + 8`, `p + 24`, `p + 40`, ..., belong to the region r_2. Given this setting, the read and write effects are as shown in Figure 6. It is now a simple exercise to show that any execution of the optimized program is equivalent to an execution of the sequential program. For instance, any instruction with a read effect on r_1 can be permuted so that it appears immediately before the following instruction with write effect r_1 on the same region r_1. This is possible because the only effect between these two instructions is a read on the other region r_2. The same is true for permuting instructions that read on r_2.