Relational Semantics for Effect-Based Program Transformations: Higher-Order Store

Nick Benton Andrew Kennedy
Microsoft Research, Cambridge
{nick,akenn}@microsoft.com

Lennart Beringer Martin Hofmann
Ludwig Maximilians Universität, Munich
{beringer,mhofmann}@tcs.ifi.lmu.de

Abstract

We give a denotational semantics to a type and effect system tracking reading and writing to global variables holding values that may include higher-order effectful functions. Refined types are modelled as partial equivalence relations over a recursively-defined domain interpreting the untyped language, with effect information interpreted in terms of the preservation of certain sets of binary relations on the store.

The semantics validates a number of effect-dependent program equivalences and can thus serve as a foundation for effect-based compiler transformations.

The definition of the semantics requires the solution of a mixed-variance equation which is not accessible to the hitherto known methods. We illustrate the difficulties with a number of small example equations one of which is still not known to have a solution.

Categories and Subject Descriptors F.3.2 [*Logic and Meanings of Programs*]: Semantics of Programming Languages – Denotational semantics, Program analysis; F.3.2 [*Logic and Meanings of Programs*]: Studies of Program Constructs – Type structure

General Terms verification, theory

Keywords type and effect systems, logical relations, domain theory, program transformation

1. Introduction

This paper is a next step in an ongoing programme developing extensional relational semantics for effect analyses which can validate effect-dependent program equivalences such as

$$x = e; y = e; e'(x, y) \quad \text{is equivalent to} \quad x = e; e'(x, x)$$

provided that e does not read from memory cells that it writes to.

Our initial work along these line [3] considered a total functional language with global integer references. We later [4] extended the language and the semantics with dynamic allocation, modelled with Kripke logical relations indexed over partial bijections. The main novelty was a semantic account of effect masking and the additional program equivalences that it validates. A subsequent tutorial presentation [8] also treated recursion, interpreting types and effects as relations over cpos, rather than just sets.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

Our next step was to try to extend the semantics to encompass dynamically allocated references to arbitrary values, including effectful functions (higher-order store). This proved too big a bite to swallow; the unexpected major obstacle being the proof of existence of a logical relation given implicitly by a mixed-variance recursive equation.

In this paper, we go a step sideways and give a denotational, relational semantics to an effect-typed functional language with higher-order store but *without* dynamic allocation. The difficulties with the existence of logical relations show up here already, but in slightly tamer form than when one also has dynamic allocation. Even so, we still find we need to restrict the precision of the effect information associated with stored functions to ensure the existence of our recursive relation.

How to solve mixed-variance recursive domain equations is well understood. There are also well-known methods for making recursive definitions of predicates and relations over such solutions [10], but these do not suffice to establish the existence of the relations we need to interpret our refined types. Other researchers have encountered related problems, which they have addressed in more or less ad hoc ways, such as Bohr's quaternary relations [6], Reus' and Schwinghammer's choice functions [11] and the 'approximate' locations of Birkedal et al [5]. In each case, the problems are circumvented by changing the implicit definition to make it fit the existing solution theory. The price to be paid in each case is a more unwieldy logical relation. E.g. quaternary rather than binary [6], depending on choice functions [6], referring to an altered semantics of the untyped language [5]

As well as describing our partial progress on denotational reasoning about higher-order store, an aim of this paper is to highlight the need for a more general solution theory of mixed-variance equations over predicates and relations. To that end, we present three small, boiled down examples of such equations including one open problem.

We should remark that a great deal of progress has recently been made in reasoning about equivalences in higher-order languages using step-indexed logical relations over operational semantics [1, 2]. It is possible that similar techniques could be used to reason about our effect-refined types although that has not been done yet. But denotational methods have their own advantages[1] and really *should* work just as well (the underlying adequate semantics one gets using a recursively-defined domain seems straighforward and elegant, and clearly contains the information we want).

We refer to our previous work [3] for more general motivation of effect-based program equivalences and their relationship to compiler correctness. The rest of this paper is organised as follows:

[1] Including being much closer to the intuitions of functional programming, understanding 'functions' as real functions.

Section 2 defines an a priori untyped functional language with global references. The references can contain arbitrary values including functional ones. We give a number of examples illustrating the use of higher-order store.

In Section 3 we give a standard denotational semantics to this language which models terms as functions from environments to computations with an environment mapping free variables to values and computations mapping stores to pairs of stores and values. Values then are defined as a solution to a recursive (pre)domain equation. In essence, a value is either an integer or a boolean or a function from values to computations. Stores are mappings from global variables to values. All functions are required to be continuous so as to enable recursive definitions.

The material of those first two sections is standard except possibly for the "no-frills" and self-contained style of presentation.

Section 4 defines a type and effect system for the language that tracks read and write access to global variables in the standard way. We do not allow the type of a reference to change upon assignments, thus we do not allow "strong updates". We note that we refrain from first defining simple types without effect information and thus define the effect typing directly for the untyped language.

Section 5 defines the equational theory that we intend to justify with a logical relation thus yielding soundness for observational equivalence. The equational theory is essentially a congruence closure of basic rules for code transformation in the presence of effect information. Importantly, the equational theory is for arbitrary types so we can deduce equations whose validity relies on the absence of certain latent effects in its functional free variables. While the formulation of the equational theory is essentially folklore and merely formalises the intuitions one might have about soundness of code transformations, we are not aware of a proof of soundness for this or a related theory by any method, be it denotational or operational.

Section 6 recalls the theory of subsets of predomains and their definition by mixed-variance recursive equations. Theorem 1 generalises the solution formula given by Pitts [10]. We work directly with retractions, in the style of Reynolds [12], rather than exploiting the more abstract category-theoretic structure of the underlying predomain; others have recently taken a similar approach, including Birkedal et al. [5]. The idea of using retractions other than the 'standard' ones arising from the construction of the recursive domain is new and allows us to establish the existence of the set of "hereditarily pure functions" in Section 6.1. Section 6 contains two more examples of mixed-variance recursive equations, one provably unsolvable and one an open problem.

Section 8 contains the recursive specification of the logical relation used to interpret types with effect information as partial equivalence relations on values. In the absence of higher-order store the specification can be read as a definition by induction on types. Here we must prove its existence using Theorem 1. We do so in Theorem 2. The remainder of the section is devoted to the proof that the logical relation indeed satisfies the equational theory (Theorems 3 and 4). The final Lemma 19 asserts that initial stores satisfy semantic store specification which in turn entails that logical relatedness of closed terms (of higher type) entails their observational equivalence.

Section 9 concludes.

Acknowledgement: Support by MS Research Cambridge (Hofmann), EU grant MOBIUS IST-FET-15905 (Beringer, Hofmann) is gratefully acknowledged.

2. Base Language

We consider an a priori untyped call-by-value lambda calculus with global references. Types will appear only later in the form of refined types delineating effects. In order to simplify the presentation we always assume A-normal form (ANF) except in concrete examples. This means that term formers are applied to variables whenever possible thus simplifying denotational semantics, typing rules and, most importantly, proofs.

The syntax of terms is then given as follows:

$$\begin{array}{rcl} e & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid x_1\ \mathit{op}\ x_2 \mid () \mid (x_1, x_2) \mid x.1 \mid \\ & & x.2 \mid x_1\ x_2 \mid \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2 \mid !\ell \mid \ell := x \mid \\ & & \texttt{if}\ x\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{rec}\ f\ x.e \mid \lambda x.e \end{array}$$

Here n ranges over integer constants, x ranges over a set of variables, ℓ ranges over a finite set $\mathbb{L}$ of locations, and op ranges over a suitable set of binary operators including arithmetic operations and comparisons.

Assignment is written as $\ell := x$, while expression $!\ell$ denotes the value contained in the location ℓ, and $\texttt{rec}\ f\ x.e$ the function f recursively defined by $f\ x = e$. The variables x, f are thus bound in this expression.

In examples, we may also use coproduct types, records, and variants with the usual SML-style syntax. The use of `ref` should be interpreted as simply setting up the initial state of global variables.

3. Denotational semantics of the untyped language

We give a denotational semantics to this language using predomains and continuous functions. A predomain is a partial order D such that whenever $(d_i)_i$ is a chain in D, i.e., $d_i \in D, d_i \sqsubseteq d_{i+1}$ for $i \geq 0$, then it has a least upper bound $\bigsqcup_i d_i \in D$. A function between predomains is continuous if it is monotone and preserves these least upper bounds. All functions between predomains are presumed continuous without explicit notice. We denote function composition by semicolon, i.e., $(f; g)(x) = g(f(x))$. We write *id* for the identity function. If D is a predomain then $D_\perp$ is $D \cup \{\perp\}$ where $\perp \notin D$ is a new least element adjoined to D.

If D_1, D_2 are disjoint predomains we write $D_1 + D_2$ for their union, viewed as a predomain with the component-wise order. If D is a predomain and a is an identifier, $a(D)$ is the predomain (isomorphic to D) whose members are expressions of the form $a(d)$ with $d \in D$. This is typically used to produce disjoint summands. For example, if D_1 and D_2 are not necessarily disjoint we can form $\mathit{inl}(D_1) + \mathit{inr}(D_2)$.

We write $D_1 \times D_2$ for the product predomain of D_1, D_2 whose elements are pairs (d_1, d_2) where $d_i \in D_i$. The order on $D_1 \times D_2$ is component-wise. We represent by $[D_1 \to D_2]$ the predomain consisting of the continuous functions from D_1 to D_2 ordered pointwise. Thus, if $f, g \in [D_1 \to D_2]$ then $f \sqsubseteq g$ means $f(x) \sqsubseteq g(x)$ for all $x \in D_1$.

A continuous function $p : D \to D$ where D is a predomain is a *retract* if $p; p = p$ and $p(x) \sqsubseteq x$. If $p : D \to D$ is a retract then we define $\mathrm{Fix}(p) = \{d \mid p(d) = d\}$. This set forms a sub-predomain of D and furthermore, we have in fact $p : D \to \mathrm{Fix}(p)$ and p together with the inclusion $\mathrm{Fix}(p) \subseteq D$ forms an embedding-projection pair. We also remark that $\mathrm{Fix}(p)$ coincides with the image of p.

Stores are then modelled as total functions from $\mathbb{L}$ to values. Since values include side-effecting functions that operate on the store, the stores and values are mutually dependent on each other leading to a recursive equation.

We thus construct a predomain of values as the least solution to the following mixed-variance predomain equation

$$\begin{array}{rcl} \mathbf{V} & \cong & \{\mathit{wrong}\} + \mathit{unit}() + \mathit{int}(\mathbb{Z}) + \mathit{bool}(\mathbb{B}) + \\ & & \mathit{pair}(\mathbf{V} \times \mathbf{V}) + \mathit{fun}(\mathbf{V} \to \mathbf{C}) \end{array}$$

where $\mathbf{C} = \mathbf{S} \to (\mathbf{S} \times \mathbf{V})_\perp$ and $\mathbf{S} = \mathbb{L} \to \mathbf{V}$ model computations and stores respectively. Note that $\mathbf{C}$ has least element $\lambda x.\perp$.

If $s \in \mathbf{S}$ and $\ell \in \mathbb{L}$ we write $s.\ell$ for $s(\ell)$ and if additionally $v \in \mathbf{V}$ then we write $s[\ell \mapsto v]$ for the store obtained from s by setting ℓ to v.

Proposition 1. *There is a family of retracts $p_i : \mathbf{V} \to \mathbf{V}$ and $p_i : \mathbf{S} \to \mathbf{S}$ and $p_i : \mathbf{C} \to \mathbf{C}$ (overloaded notation) for $i \in \mathbb{N}$ obeying the following equations:*

$$\begin{array}{rcl} p_i(\mathit{wrong}) & = & \mathit{wrong} \\ p_i(\mathit{int}(n)) & = & \mathit{int}(n) \\ p_i(\mathit{unit}()) & = & \mathit{unit}() \\ p_i(\mathit{bool}(x)) & = & \mathit{bool}(x) \\ p_i(\mathit{pair}(v_1, v_2)) & = & \mathit{pair}(p_i(v_1), p_i(v_2)) \\ p_i(\mathit{fun}(g)) & = & \mathit{fun}(p_i; g; p_i) \\ p_0(f)(s) & = & \perp \\ p_{i+1}(f)(s) & = & \perp \textit{ if } f(p_i(s)) = \perp \\ p_{i+1}(f)(s) & = & (p_i(s_1), p_i(v)) \textit{ if } f(p_i(s)) = (s_1, v) \\ p_i(s)(\ell) & = & p_i(s(\ell)) \end{array}$$

Moreover, $p_i \sqsubseteq p_{i+1}$ and $p_i; p_j = p_{\min(i,j)}$ and $\bigsqcup_i p_i(x) = x$ for all $x \in \mathbf{V} \cup \mathbf{S} \cup \mathbf{C}$.

Sketch. We define the p_i recursively by the given equations. Let π_i be the family of the standard partial retractions derived from the construction of $\mathbf{V}$. We can show by induction on i that $p_i(x) \sqsupseteq \pi_j(x)$ where j is the least integer above i such that $\pi_j(x) \neq \perp$. □

An *environment* is a finite map from variables to elements of $\mathbf{V}$. The environments form a predomain Env by putting

$$\theta \sqsubseteq \theta' \iff \mathrm{dom}(\theta) = \mathrm{dom}(\theta') \wedge \forall x{:}\mathrm{dom}(\theta).\theta(x) \sqsubseteq \theta'(x)$$

To each term e we assign a continuous function $[\![e]\!] : \mathrm{Env} \to \mathbf{C}$ by the clauses in Fig. 1, where it is assumed that e is in ANF. We may abbreviate $[\![e]\!]\emptyset$ or indeed $[\![e]\!]\theta$ for arbitrary θ by just $[\![e]\!]$ when e is closed.

4. Type and effect system

We now present our effect analysis that, in addition to statically typing terms in the usual way (and so preventing *wrong* being the result of any well-typed computation), uses effect annotations to track the dependency and effect on the store of computations. Note that our effect systems does not track non-termination, which may be introduced by recursion either directly or through the store.

4.1 Types

Effects, ranged over by ε, are subsets of the set of *elementary effects*:$\{rd_\ell, wr_\ell \mid \ell \in \mathbb{L}\}$. We tend to sometimes omit set braces thus writing rd_ℓ for $\{rd_\ell\}$. The types are given by the following grammar:

$$A, B, C \quad ::= \quad \mathtt{int} \mid \mathtt{unit} \mid \mathtt{bool} \mid A \times B \mid A \xrightarrow{\varepsilon} B$$

Given a finite set of locations Π we write $\Pi \vdash A\ ok$ to mean that all locations occurring in (effect annotations within) A are contained in Π. We define $\Pi \vdash \varepsilon\ ok$ analogously. A typing context Θ is a mapping from a finite set of variables $\mathrm{dom}(\Theta)$ to types. We write $\Pi \vdash \Theta\ ok$ to mean that $\Pi \vdash \Theta(x)\ ok$ for all $x \in \mathrm{dom}(\Theta)$.

For ε an effect we define $\mathrm{rds}(\varepsilon)$ as the set of locations ℓ such that $rd_\ell \in \varepsilon$. Likewise, $\mathrm{wrs}(\varepsilon) = \{\ell \mid wr_\ell \in \varepsilon\}$. We write $\mathrm{locs}(\varepsilon) = \mathrm{rds}(\varepsilon) \cup \mathrm{wrs}(\varepsilon)$. An effect ε is *storable* if $\mathrm{rds}(\varepsilon) = \mathrm{wrs}(\varepsilon) (= \mathrm{locs}(\varepsilon))$. A type is storable if all effects occurring in it are storable.

Given a finite set Π of locations, a *store type* (for Π) is a finite mapping Σ assigning to each $\ell \in \Pi$ a *storable* type $\Sigma(\ell)$ such that $\Pi \vdash \Sigma(\ell)\ ok$. We may write a concrete store typing for $\{\ell_1, \ldots, \ell_n\}$ in the form $\ell_1{:}A_1, \ldots \ell_n{:}A_n$ with $A_i = \Sigma(\ell_i)$. The intention is that given a store typing Σ for Π then a location $\ell \in \Pi$ is supposed to contain values of type $\Sigma(\ell)$. Only storable types are permitted in this position.

For example,

$$\Sigma \equiv \ell_1{:}\mathtt{int}, \ell_2{:}\mathtt{unit} \xrightarrow{rd_{\ell_2}, wr_{\ell_2}} \mathtt{unit}$$

is a store typing which prescribes that ℓ_1 holds integers, whereas location ℓ_2 contains commands which may access ℓ_2 itself in an arbitrary way.

Now the type $\mathtt{unit} \xrightarrow{rd_{\ell_1}, rd_{\ell_2}} \mathtt{unit}$ contains commands that may read from ℓ_1, ℓ_2. This type is not storable but may nevertheless appear as the type of terms or variables. Storability is a technical condition which we currently need in order to establish the existence of the semantic meaning of types. We will discuss its implications and possible relaxations later.

```
val r = ref (fn x => 0);
val f = (r := (fn x => if x=0 then 1
                       else x * (!r) (x-1));
         !r);
```

Figure 2. Factorial by backpatching

4.2 Initial store

Languages with global store always undergo the annoying difficulty of what the contents of an uninitialised location should be. Unfortunately, in our denotational model there is no single value that could serve as initialisation without compromising any reasonable definition of type soundness.

Therefore, we now define for each type A a default value v_A by

$$\begin{array}{rcl} v_{\mathtt{unit}} & = & () \\ v_{\mathtt{bool}} & = & \mathit{bool}(\mathtt{true}) \\ v_{\mathtt{int}} & = & \mathit{int}(0) \\ v_{A \times B} & = & \mathit{pair}(v_A, v_B) \\ v_{A \xrightarrow{\varepsilon} B} & = & \mathit{fun}(\lambda x.\lambda s.\perp) \end{array}$$

If Σ is a store typing we define the initial store s_Σ by $s_\Sigma.\ell = v_{\Sigma(\ell)}$ for $\ell \in \mathrm{dom}(\Sigma)$ and $s_\Sigma.\ell = \mathit{wrong}$ otherwise.

4.3 Typing rules

The *typing judgement* has the form $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$ with Π a finite set of locations, Σ a store typing, Θ a typing context, A a type, ε an effect. Such a typing judgement is *well formed* if $\Pi \vdash \Sigma\ ok$ and $\Pi \vdash \Theta\ ok$ and $\Pi \vdash A\ ok$ and $\Pi \vdash \varepsilon\ ok$. Whenever a typing judgement appears its wellformedness is assumed and mentioned explicitly only if needed for clarity.

We may abbreviate $\Pi; \Sigma; \Theta \vdash e : A, \emptyset$ by $\Pi; \Sigma; \Theta \vdash e : A$. We also abbreviate $A \xrightarrow{\emptyset} B$ by just $A \to B$. The typing judgement is defined in Fig. 5.

4.4 Examples

The following examples illustrate our effect system, and the restrictiveness of the storability condition in particular. For the sake of readability we use ML-style syntax.

Factorial by backpatching Figure 2 presents a program for computing the factorial function, where the recursion is achieved using a reference. For $\varepsilon = \{rd_{\mathtt{r}}, wr_{\mathtt{r}}\}$, we have

$$\mathtt{r}; \mathtt{r} : \mathtt{int} \xrightarrow{\varepsilon} \mathtt{int}; \emptyset \vdash \mathtt{f} : \mathtt{int} \xrightarrow{\varepsilon} \mathtt{int}.$$

This typing is less precise than the more intuitive

$$\mathtt{r}; \mathtt{r} : \mathtt{int} \xrightarrow{rd_{\mathtt{r}}} \mathtt{int}; \emptyset \vdash \mathtt{f} : \mathtt{int} \xrightarrow{rd_{\mathtt{r}}, wr_{\mathtt{r}}} \mathtt{int}.$$

$$
\begin{array}{lcl}
[\![x]\!]\theta\ s & = & (s, \theta(x)) \\
[\![c]\!]\theta\ s & = & (s, \mathit{bool}(c)) \\
[\![x\ \mathit{opy}]\!]\theta\ s & = & \theta(x)\ \mathit{op}\ \theta(y) \\
[\![(x,y)]\!]\theta\ s & = & (s, \mathit{pair}(\theta(x), \theta(y))) \\
[\![\mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2]\!]\theta\ s & = & [\![e_2]\!]\theta[x \mapsto v]\ s_1 \text{when } [\![e_1]\!]\theta\ s = (s_1, v) \\
[\![\mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2]\!]\theta\ s & = & \bot, \text{ when } [\![e_1]\!]\theta\ s = \bot \\
[\![\mathtt{if}\ x\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]\theta & = & [\![e_2]\!]\theta, \text{ when } \theta(x) = \mathit{bool}(\mathtt{true}) \\
[\![\mathtt{if}\ x\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]\theta & = & [\![e_3]\!]\theta, \text{ when } \theta(x) = \mathit{bool}(\mathtt{false}) \\
[\![\mathtt{rec}\ f\ x.e]\!]\theta\ s & = & (s, \mathit{fun}(g)) \text{ where } g = \bigsqcup_i g_i \text{ and} \\
 & & g_0 = \lambda x.\lambda s.\bot \text{ and} \\
 & & g_{i+1} = \lambda v.[\![e]\!]\theta[x \mapsto v, f \mapsto \mathit{fun}(g_i)]
\end{array}
$$

$$
\begin{array}{lcl}
[\![n]\!]\theta\ s & = & (s, \mathit{int}(n)) \\
[\![()]\!]\theta\ s & = & (s, \mathit{unit}()) \\
 & & \\
[\![x\ y]\!]\theta\ s & = & f(\theta(y))s \text{ where } \theta(x) = \mathit{fun}(f) \\
[\![x.1]\!]\theta\ s & = & (s, v_1) \text{ if } \theta(x) = \mathit{pair}(v_1, v_2) \\
[\![x.2]\!]\theta\ s & = & (s, v_2) \text{ if } \theta(x) = \mathit{pair}(v_1, v_2) \\
[\![!\ell]\!]\theta\ s & = & (s, s.\ell) \\
[\![\ell := y]\!]\theta\ s & = & (s[\ell \mapsto \theta(y)], \mathit{unit}()) \\
[\![\lambda x.e]\!]\theta\ s & = & (s, \mathit{fun}(f)) \text{ where } f\ v = [\![e]\!]\theta[x \mapsto v] \\
[\![e]\!]\theta\ s & = & \mathit{wrong}, \text{ if none of these clauses applies}
\end{array}
$$

Figure 1. Denotational Semantics

```
(* Globals *)
val v = ref (Vector.fromList [0]);
val w = ref (Vector.fromList [0]);
val prog = ref (fn () => ());
val res = ref 0;

fun comp g = let val f = !prog
             in prog := (fn x => g (f x)) end;

fun template elem m =
  fn () => res := !res + elem * Vector.sub(!w,m);

fun preeval () =
  let val n = Vector.length (!v)
      val counter = 0
  in prog := (fn () => res :=0); loop counter n
  end
and next counter n = loop (counter + 1) n
and loop counter n =
  if counter = n then !prog else
  let val elem = Vector.sub(!v,counter)
  in if elem = 0 then next counter n else
     (comp (template elem counter);
      next counter n)
  end;

fun apply p x = (w := Vector.fromList x; p(); !res);

fun main vval wvals =
  (v:=Vector.fromList vval;
  let val p = preeval() in map (apply p) wvals end);
```

Figure 3. Vector multiplication

In particular, the latter typing indicates that the content of `r` is commutative in the sense that it may be subject to the transformation rule E-SWAP below. However, this typing is not well-formed, since the type of `r` is not storable. We suspect that operational techniques may be used to show commutativity of this particular example, though not necessarily of arbitrary applications of backpatching. Indeed, it is unclear whether the more precise typing is semantically meaningful at all, as the *reading-but-not-writing* pattern exhibited by the optimised typing of `r` is precisely at the heart of the *hereditary read-only* challenge we set in Section 6.2.

Vector multiplication The code in Figure 3 performs integer vector mutiplications $\mathtt{vvals} * \mathtt{wvals}_1, \ldots, \mathtt{vvals} * \mathtt{wvals}_n$ using vector[2] `v` and `w`. Inspired by the self-modifying assembly-level code for the same problem from [7], the program is partitioned into two phases. The first phase (function `preeval`) constructs optimised code in reference `prog`, by iterating over vector `v`. Each non-zero entry of `v` results in the (partially) generated program being extended by the appropriate instantiation of the `template` instruction. Function `apply` executes the program retrieved from `prog` on its second argument, assigned to vector `w`. The `main` function assigns the integer list `vvals` to `v`, triggers pre-evaluation, and then maps the resulting program over the inputs $\mathtt{wvals}_i$ by invoking `apply` for each element in turn.

The constructed program, when executed, calculates the result in the reference `res`, and only accesses `w`. In contrast, vector `v` is only accessed during the pre-evaluation.

The functions would be typed as follows in our system:

$$\{\mathtt{res}, \mathtt{w}\}; \Sigma_{\mathtt{templ}}; \emptyset \vdash \mathtt{template} : \mathtt{int} \to \mathtt{int} \to A$$
$$\Pi; \Sigma; \emptyset \vdash \mathtt{preeval} : \mathtt{unit} \xrightarrow{\varepsilon_2} A$$
$$\Pi; \Sigma; \emptyset \vdash \mathtt{next} : \mathtt{int} \to \mathtt{int} \xrightarrow{\varepsilon_2} A$$
$$\Pi; \Sigma; \emptyset \vdash \mathtt{loop} : \mathtt{int} \to \mathtt{int} \xrightarrow{\varepsilon_2} A$$
$$\{\mathtt{res}, \mathtt{w}\}; \Sigma_{\mathtt{templ}}; \emptyset \vdash \mathtt{apply} : A \to \mathtt{int\ list} \xrightarrow{\varepsilon_1} \mathtt{int}$$
$$\Pi; \Sigma; \emptyset \vdash \mathtt{main} : \mathtt{int\ list} \to \mathtt{int\ list\ list} \xrightarrow{\varepsilon} \mathtt{int\ list}$$

where

$$
\begin{array}{rcl}
\Sigma_{\mathtt{templ}} & = & \mathtt{res} : \mathtt{int}, \mathtt{w} : \mathtt{int\ array} \\
\Pi & = & \{\mathtt{res}, \mathtt{v}, \mathtt{w}, \mathtt{prog}\} \\
\Sigma & = & \Sigma_{\mathtt{templ}}, \mathtt{prog} : A, \mathtt{v} : \mathtt{int\ array} \\
A & = & \mathtt{unit} \xrightarrow{\varepsilon_1} \mathtt{unit} \\
\varepsilon_1 & = & \{rd_{\mathtt{res}}, wr_{\mathtt{res}}, rd_{\mathtt{w}}, wr_{\mathtt{w}}\} \\
\varepsilon_2 & = & \{rd_{\mathtt{v}}, rd_{\mathtt{prog}}, wr_{\mathtt{prog}}\} \\
\varepsilon & = & \{wr_{\mathtt{v}}\} \cup \varepsilon_1 \cup \varepsilon_2
\end{array}
$$

Despite the fact that `w` is only assigned to in `apply`, the effect $wr_{\mathtt{w}}$ is included in ε_1 and thus occurs wherever A does. This is again required by the storability condition because A occurs as the type of the (function) reference `prog`. In contrast to the previous example, there is no circular relationship between the location the effect concerns, `w`, and the location with which the type containing the effect is associated, `prog`. Indeed, we believe the existence proof of the logical relation in Section 8 could be generalised to a more relaxed notion of storability, where (sets of) locations are hierarchically ordered in stages, and that this more permissive notion would admit the deletion of $wr_{\mathtt{w}}$ from ε_1 in A and only be

[2] We assume that all lists are of equal length. For any fixed length the use of vectors may in fact be considered syntactic sugar as we may represent vectors by appropriately nested products.

```
fun execute () =
  let val (i,n) = getevent()
      val x = if i = 1 then (!e1) n else
              if i = 2 then (!e2) n else
                :
              if i = k then (!ek) n else (!e0) n
  in (output x) end;
```

Figure 4. Event handling

added in the effect of `apply` via

$$\{\mathtt{res}, \mathtt{w}\}; \Sigma_{\mathtt{templ}}; \emptyset \vdash \mathtt{apply} : A \to \mathtt{int\ list} \xrightarrow{wr_{\mathtt{w}} \cup \varepsilon_1} \mathtt{int}.$$

We leave the proof of this conjecture as future work. Note that it is not necessary to include $wr_{\mathtt{v}}$ in ε_2, since ε_2 does not occur as the effect of a reference-held function.

Event handling Figure 4 presents code for a simple event handler, where the environmental function `getevent` generates pairs comprising a handler identifier and an argument that is passed on to the selected handler, and `output` displays the result of the chosen handler. The code for the handlers is located in references `e_0`, ... `e_k`. For

$$\begin{aligned}\Sigma \quad = \quad & \mathtt{getevent} :: \mathtt{unit} \to \mathtt{int} \times \mathtt{int}, \\ & \mathtt{output} : \mathtt{int} \to \mathtt{unit}, \mathtt{e\_i} : \mathtt{int} \xrightarrow{\varepsilon_i} \mathtt{int},\end{aligned}$$

$\Pi = \mathrm{dom}(\Sigma)$, and $\varepsilon = \cup_{i=0,\ldots,k}\{\varepsilon_i, rd_{\mathtt{e}_i}\}$ we have

$$\Pi; \Sigma; \emptyset \vdash \mathtt{execute} : \mathtt{unit} \xrightarrow{\varepsilon} \mathtt{unit},$$

and may now consider policies restricting the mutual accessibility of handlers. For example, $\varepsilon_1 = \emptyset$ requires the first handler to be pure, while $\varepsilon_2 = \{rd_{\mathtt{e}_3}, wr_{\mathtt{e}_3}\}$ means that the second handler may read (hence execute) as well as overwrite the third handler, and $\varepsilon_3 = \{rd_{\mathtt{e}_3}, wr_{\mathtt{e}_3}\}$ indicates that the latter is self-modifying.

Again, the storability condition is rather strict. For example, it requires that if we want to give a handler read access to some other handler (for example for executing it as part of its own event handling), we also have to give it write permission to this other handler. As further motivation for a relaxed notion of storability, one may consider the following situation for $k = 4$.

$$\begin{aligned}\varepsilon_1 \quad &= \quad \emptyset \\ \varepsilon_2 \quad &= \quad \{rd_{\mathtt{e}_1}\} \\ \varepsilon_3 \quad &= \quad \{rd_{\mathtt{e}_2}\} \\ \varepsilon_4 \quad &= \quad \{rd_{\mathtt{e}_2}, wr_{\mathtt{e}_0}\} \\ \varepsilon_0 \quad &= \quad \{rd_{\mathtt{e}_2}, wr_{\mathtt{e}_0}, rd_{\mathtt{e}_1}\}\end{aligned}$$

Here, handler 1 is pure, handler 2 may read handler 1 (and execute it, since $\varepsilon_1 \subseteq \varepsilon_2$), and handler 3 may read handler 2, *but not execute it*, since executing it might involve reading from $\mathtt{e}_1$, which is not permitted by ε_3. Since handler 3 can't write any handler it is doubtful handler 3 can do anything useful with its access permission to handler 2. Handlers 4 and 0 repair the deficiency of handler 3. The former allows $\mathtt{e}_0$ to be modified, so handler 4 could, for example, copy $\mathtt{e}_2$'s content to $\mathtt{e}_0$ and then return its own argument (still handler 4 cannot *execute* `e_2`). Finally, the default handler 0 is allowed to read `e_2` *and execute its content* (since it also has the $rd_{\mathtt{e}_1}$ permission), and also to modify itself, for example by overwriting itself with `e_2`'s content. Note that the permission that `e_4` and `e_0` may use their write permission to `e_0` to copy the content of `e_2` to `e_0` is sanctioned by $\varepsilon_2 \subseteq \varepsilon_0$.

5. Equational theory

We are interested in justifying the equational theory given in Fig 6. The theory is essentially the same as that of our earlier work [3], with the important difference that references now hold values of higher type and hence with nontrivial equality. By "justifying" we mean the construction of a natural semantic model that validates the equations and has the property of entailing observational equivalence.

We note the absence of a rule for dead computation yielding $e = ()$ provided $e : \mathtt{unit}, \varepsilon$ where $\mathrm{wrs}(\varepsilon) = \emptyset$. Such a rule is not sound in the presence of non-termination and for a similar reason rule E-HOIST has a dummy evaluation of the code to be hoisted. The other rules are standard. We remark that rule E-BASIC together with E-TRANS subsumes all kinds of congruence and $\beta\eta$-like rules.

In future work we could develop an inequational theory approximating observational approximation rather than equivalence. This would lead to a more natural version of hoisting allowing us to reinstate "dead code" in the form $e \sqsubseteq () : \mathtt{unit}, \varepsilon$.

6. Admissible predicates and relations

A subset $P \subseteq D$ of a predomain D is *admissible* if whenever $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$ is an ascending chain with $d_i \in P$ for all i then $\bigsqcup_i d_i \in P$, too. The admissible subsets of D are denoted by $\mathrm{Adm}(D)$.

In particular, a relation $R \subseteq D \times D$ is admissible if it is admissible qua subset of the product predomain $D \times D$.

A binary relation on a set is a *partial equivalence relation* (PER) if it is symmetric and transitive. The set of PERs on a set is closed under arbitrary intersections and disjoint unions. The same is true for admissible PERs on a predomain.

Definition 2. *If D is a predomain and $R \subseteq D \times D$ then $\mathrm{per}(R)$ is the least admissible PER containing R.*

Proposition 3. *Let D_1, D_2, D be predomains and $R_i \subseteq D_i \times D_i$ be relations, and $R \subseteq D \times D$ an admissible PER.*

1. $\mathrm{per}(R_1 \times R_2) = \mathrm{per}(R_1) \times \mathrm{per}(R_2)$.
2. *If $f : D_1 \to D$ is continuous then $f^{-1}R$ (component-wise) is an admissible PER on D_1.*
3. *If $g : D_1 \times D_2 \to D$ is continuous and $g \times g : R_1 \times R_2 \to R$ then $g \times g : \mathrm{per}(R_1) \times \mathrm{per}(R_2) \to R$.*

Proof. The "$\subseteq$" direction of the first item is obvious. For the converse consider $\{(x, x') \mid \forall (y, y') \in \mathrm{per}(R_2).\mathrm{per}(R_1 \times R_2)((x, x'), (y, y'))\}$. It is easy to see that this relation is an admissible PER comprising R_1 and the claim follows.

The second item is obvious, the third one is a corollary of the first two. □

Definition 4. *An* inductive presentation *of predomain D is given by an increasing family of retracts $(p_i)_i$ on D such that $\bigsqcup_i p_i(x) = x$ for all $x \in D$.*

For example, the product $\mathbf{V} \times \mathbf{V}$ forms a predomain with inductive presentation $p_i(v, v') = (p_i(v), p_i(v'))$.

Definition 5. *Let $(p_i)_i$ be an inductive presentation of predomain D. A subset $P \subseteq D$ is* uniform *(with respect to the presentation) if $x \in P$ implies $p_i(x) \in P$ for all i.*

The following theorem allows us to define admissible subsets of predomains as solutions of certain mixed-variance equations. While clearly inspired by the existing solution theory for such equations [10] it goes beyond the latter by not insisting that the approximating sets be uniform with respect to the standard inductive presentation (the p_i or tuples thereof).

$$\frac{}{\Pi;\Sigma;\Theta \vdash n : \mathtt{int}}(\text{T-INT}) \qquad \frac{x \in \mathrm{dom}(\Theta)}{\Pi;\Sigma;\Theta \vdash x : \Theta(x)}(\text{T-VAR})$$

$$\frac{}{\Pi;\Sigma;\Theta \vdash !\ell : \Sigma(\ell), \{rd_\ell\}}(\text{T-READ}) \qquad \frac{\Pi;\Sigma;\Theta \vdash y : \Sigma(\ell)}{\Pi;\Sigma;\Theta \vdash \ell := y : \mathtt{unit}, \{wr_\ell\}}(\text{T-WRITE})$$

$$\frac{\Pi;\Sigma;\Theta \vdash e : A, \varepsilon_1 \qquad A <: B \qquad \varepsilon_1 \subseteq \varepsilon_2}{\Pi;\Sigma;\Theta \vdash e : B, \varepsilon_2}(\text{T-SUB}) \qquad \frac{\Pi;\Sigma;\Theta \vdash x : A \xrightarrow{\varepsilon} B \qquad \Pi;\Sigma;\Theta \vdash y : A}{\Pi;\Sigma;\Theta \vdash x\ y : B, \varepsilon}(\text{T-APP})$$

$$\frac{\Pi;\Sigma;\Theta, x{:}A \vdash e : B, \varepsilon}{\Pi;\Sigma;\Theta \vdash \lambda x.e : A \xrightarrow{\varepsilon} B}(\text{T-LAM}) \qquad \frac{\Pi;\Sigma;\Theta \vdash x : \mathtt{bool} \qquad \Pi;\Sigma;\Theta \vdash e_1 : A, \varepsilon \qquad \Pi;\Sigma;\Theta \vdash e_2 : A, \varepsilon}{\Pi;\Sigma;\Theta \vdash \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : A, \varepsilon}(\text{T-IF})$$

$$\frac{\Pi;\Sigma;\Theta \vdash e_1 : A_1, \varepsilon_1 \qquad \Pi;\Sigma;\Theta, x{:}A_1 \vdash e_2 : A_2, \varepsilon_2}{\Pi;\Sigma;\Theta \vdash \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2 : A_2, \varepsilon_1 \cup \varepsilon_2}(\text{T-LET}) \qquad \frac{\Pi;\Sigma;\Theta \vdash x : A \qquad \Pi;\Sigma;\Theta \vdash y : B}{\Pi;\Sigma;\Theta \vdash (x, y) : A \times B}(\text{T-PAIR})$$

$$\frac{\Pi;\Sigma;\Theta, f{:}A \xrightarrow{\varepsilon} B, x{:}A \vdash e : B, \varepsilon}{\Pi;\Sigma;\Theta \vdash \mathtt{rec}\ f\ x.e : A \xrightarrow{\varepsilon} B}(\text{T-REC}) \qquad \frac{}{A <: A}(\text{S-REFL})$$

$$\frac{A_1 <: A_2 \qquad B_1 <: B_2}{A_1 \times B_1 <: A_2 \times B_2}(\text{S-PROD}) \qquad \frac{A_2 <: A_1 \qquad B_1 <: B_2 \qquad \varepsilon_1 \subseteq \varepsilon_2}{A_1 \xrightarrow{\varepsilon_1} B_1 <: A_2 \xrightarrow{\varepsilon_2} B_2}(\text{S-ARR})$$

Figure 5. Rules for effect typing

$$\frac{\forall\theta.[\![e_1]\!]\theta = [\![e_2]\!]\theta \qquad \Pi;\Sigma;\Theta \vdash e_i : A, \varepsilon}{\Pi;\Sigma;\Theta \vdash e_1 = e_2 : A, \varepsilon}(\text{E-BASIC}) \qquad \frac{\Pi;\Sigma;\Theta \vdash e_1 = e_2 : A, \varepsilon}{\Pi;\Sigma;\Theta \vdash e_2 = e_1 : A, \varepsilon}(\text{E-SYM})$$

$$\frac{\Pi;\Sigma;\Theta \vdash e_1 = e_2 : A, \varepsilon \qquad \Pi;\Sigma;\Theta \vdash e_2 = e_3 : A, \varepsilon}{\Pi;\Sigma;\Theta \vdash e_1 = e_3 : A, \varepsilon}(\text{E-TRANS})$$

$$\frac{\Pi;\Sigma;\Theta \vdash e : A, \varepsilon \qquad \mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset \qquad x \notin \mathrm{dom}(\Theta)}{\Pi;\Sigma;\Theta \vdash \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ \mathit{pair}(x, x) = \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ \mathit{pair}(x, y) : A \times A, \varepsilon}(\text{E-DUP})$$

$$\frac{\Pi;\Sigma;\Theta \vdash e_i : A_i, \varepsilon_i \qquad \forall i = 1, 2.\mathrm{rds}(\varepsilon_i) \cap \mathrm{wrs}(\varepsilon_{3-i}) = \mathrm{wrs}(\varepsilon_i) \cap \mathrm{wrs}(\varepsilon_{3-i}) = x_i \cap (\mathrm{dom}(\Theta) \cup \{x_{3-i}\}) = \emptyset}{\begin{array}{l}\Pi;\Sigma;\Theta \vdash \mathtt{let}\ x_1 \Leftarrow e_1\ \mathtt{in}\ \mathtt{let}\ x_2 \Leftarrow e_2\ \mathtt{in}\ \mathit{pair}(x_1, x_2) = \\ \mathtt{let}\ x_2 \Leftarrow e_2\ \mathtt{in}\ \mathtt{let}\ x_1 \Leftarrow e_1\ \mathtt{in}\ \mathit{pair}(x_1, x_2) : A_1 \times A_2, \varepsilon_1 \cup \varepsilon_2\end{array}}(\text{E-SWAP})$$

$$\frac{\Pi;\Sigma;\Theta \vdash e_1 : A, \emptyset \qquad \Pi;\Sigma;\Theta, x{:}A, y{:}B \vdash e_2 : C, \varepsilon \qquad x \neq y}{\Pi;\Sigma;\Theta \vdash \mathtt{let}\ \_ \Leftarrow e_1\ \mathtt{in}\ \lambda y{:}B.\mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ e_2 = \mathtt{let}\ x \Leftarrow e_1\ \mathtt{in}\ \lambda y{:}B.e_2 : B \xrightarrow{\varepsilon} C, \emptyset}(\text{E-HOIST})$$

Figure 6. Theory of effect-dependent program equivalences

Theorem 1. *Let D be a predomain. Let $\mathcal{V}$ be a subset of the admissible subsets of D which is closed under intersections and let $V_0^+, V_0^- \in \mathcal{V}$. Let $F : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ be antitone in its first and monotone in its second argument, i.e., $U \subseteq U'$ and $V \subseteq V'$ implies $F(U', V) \subseteq F(U, V')$.*

Let $q_i : D \to D$ be a family of maps (in general but not necessarily continuous). Suppose that the following conditions are satisfied:

- *$\bigsqcup_i q_i(x) = x$ for all $x \in D$, in particular $q_i(x) \sqsubseteq q_{i+1}(x)$ for all i;*
- *$V_0^+ \subseteq F(U, V) \subseteq V_0^-$ for any $U, V \in \mathcal{V}$;*
- *$q_0 : V_0^- \to V_0^+$;*
- *If $U, V \in \mathcal{V}$ and $U \subseteq V$ and $q_i : V \to U$ then $q_{i+1} : F(U, V) \to F(V, U)$.*

Then there exists a unique $P \in \mathcal{V}$ satisfying $P = F(P, P)$.

Proof. For $i > 0$ define $V_i^+, V_i^- \in \mathcal{V}$ by $V_{i+1}^+ = F(V_i^-, V_i^+)$ and $V_{i+1}^- = F(V_i^+, V_i^-)$. The following are easily proved by induction on i and j.

- $V_0^+ \subseteq \cdots \subseteq V_i^+ \subseteq V_{i+1}^+ \subseteq \cdots \subseteq V_{j+1}^- \subseteq V_j^- \subseteq V_0^-$;
- $q_i : V_i^- \to V_i^+$

Now define $V_\infty^- = \bigcap_i V_i^- = \{d \mid \forall i.d \in V_i^-\}$. Note that $V_\infty^- \in \mathcal{V}$ as $\mathcal{V}$ is closed under intersection.

We have $V_i^+ \subseteq V_\infty^-$ and $q_i : V_\infty^- \to V_i^+$ for all i. We now claim that $F(V_\infty^-, V_\infty^-) = V_\infty^-$.

First, $F(V_\infty^-, V_\infty^-) \subseteq F(V_i^+, V_i^-) = V_{i+1}^-$ for each i, hence $F(V_\infty^-, V_\infty^-) \subseteq V_\infty^-$.

For the converse assume that $x \in V_\infty^-$. For each i we have $q_{i+1}(x) \in V_{i+1}^+ = F(V_i^-, V_i^+) \subseteq F(V_\infty^-, V_\infty^-)$. But $x = \bigsqcup_i q_i(x)$, so $x \in F(V_\infty^-, V_\infty^-)$ by admissibility.

Now suppose that $F(P, P) = P$ and $P \in \mathcal{V}$. By induction on i we get $V_i^+ \subseteq V_i^-$, thus $P \subseteq V_\infty^-$. On the other hand, if $x \in V_\infty^-$ then $q_i(x) \in V_i^+ \subseteq P$ for all i, hence $x \in P$ by admissibility. Thus, $P = V_\infty^-$. □

The following direct lemma shows one way to discharge the assumption $\bigsqcup_i q_i(x) = x$.

Lemma 6. *Suppose that D is a predomain with inductive presentation $(p_i)_i$. Suppose that for each $x \in D$ and $i \geq 0$ we have an element $q_i(x) \in D$ such that $q_i(x) \sqsubseteq q_{i+1}(x) \sqsubseteq x$ holds for all i. If, in addition, $p_i(x) \sqsubseteq q_i(x)$ then $\bigsqcup_i q_i(x) = x$.*

We note that the standard method for solving mixed-variance equations is the special case where $q_i = p_i$. Whichever family q_i we choose, the (unique) solution $P = F(P,P)$ is closed under the q_i, i.e., one has $q_i(v) \in P$ whenever $v \in P$. This is because if $v \in P$ then $v \in V_i^-$, hence $q_i(v) \in V_i^+ \subseteq P$. Therefore, in those cases where any P such that $P = F(P,P)$ cannot possibly be closed under the standard retractions p_i a solution cannot be obtained by the standard method.

The rest of this section presents some examples illustrating the kind of technicalities that arise when trying to make recursive definitions of predicates and relations over recursively-defined domains. We will return to the semantic interpretation of our particular language in Section 8.

6.1 Example: hereditarily pure functions

We now give a simple example of such a situation. Suppose that D is the solution to the domain equation $D \cong D \times D \to (D \times D)_\perp$. This predomain has the standard inductive presentation given by

$$\begin{array}{rcl} p_0(f)(s,x) & = & \perp \\ p_{i+1}(f)(s,x) & = & \perp, \text{ if } f(p_i(s),p_i(x)) = \perp \\ p_{i+1}(f)(s,x) & = & (p_i(s_1),p_i(y)), \text{ if} \\ & & \qquad f(p_i(s),p_i(x)) = (s_1,y) \end{array}$$

We think of an element $f \in D \times D \to (D \times D)_\perp$ as an untyped function side-effecting an untyped store. For the sake of the example, we are interested in the "hereditarily pure" functions, i.e., those that given a hereditarily pure argument do not depend upon nor modify the store and return a hereditarily pure result. We thus seek an admissible subset $P \subseteq D$ satisfying

$$f \in P \iff \forall x \in P.\ (\forall s \in D. f(s,x) = \perp) \vee (\exists u \in P. \forall s \in D. f(s,x) = (s,u))$$

In order to construct such P we invoke Theorem 1 with

$$f \in F(U,V) \iff \forall x \in U.\ (\forall s \in D. f(s,x) = \perp) \vee (\exists u \in V. \forall s \in D. f(s,x) = (s,u))$$

and the functions q_i given by

$$\begin{array}{rcl} q_0(f)(s,x) & = & \perp \\ q_{i+1}(f)(s,x) & = & \perp, \text{ if } f(s,q_i(x)) = \perp \\ q_{i+1}(f)(s,x) & = & (s_1,q_i(y)), \text{ if } f(s,q_i(x)) = (s_1,y) \end{array}$$

Furthermore, we put $V_0^+ = \{\lambda p.\perp\}$ and $V_0^- = D$. The premises of Theorem 1 are now easily verified thus furnishing the existence of the desired subset P. Notice that P is closed under the q_i by construction but cannot possibly be closed under the p_i. Namely, since $\lambda p.p \in P$ so would be $p_1(\lambda p.p)$, but $p_1(\lambda p.p)(s,x) = (\lambda p.\perp, \lambda p.\perp)$ which is not hereditarily pure since the store is modified from s to $\lambda p.\perp$. Notice that $q_1(\lambda p.p)(s,x) = (s, q_1(x))$.

6.2 Challenge: hereditarily read-only commands

The following open problem may illustrate why we had to impose the restriction that locations store values of storable type only.

Suppose that D is the standard solution to the predomain equation $D \cong D \to D_\perp$. This predomain has the standard inductive presentation given by

$$\begin{array}{rcl} p_0(f)(x) & = & \perp \\ p_{i+1}(f)(x) & = & \perp, \text{ if } f(p_i(x)) = \perp \\ p_{i+1}(f)(x) & = & p_i(y), \text{ if } f(p_i(x)) = y \end{array}$$

We think of an element $f \in D \to D_\perp$ as a command operating on a store with a single untyped global variable. We are interested in delineating the "hereditarily read only" commands, i.e., those that do not modify this store but whose termination behaviour may well depend upon the contents of the store. All this under the assumption that the contents of the store are themselves "hereditarily read-only".

We thus seek an admissible subset (for simplicity we only treat the unary case here) $P \subseteq D$ satisfying

$$f \in P \iff \forall x \in P. f\ x \in \{x, \perp\}$$

The problem with this definition is that $\lambda x.x \in P$ yet $p_i(\lambda x.x) \notin P$ unless $i = 0$ and we cannot see how this could be fixed if we replaced p_i by some other canonical family of retractions. Yet, the solutions obtained by the hitherto available methods are always closed under some reasonable family of retractions, typically, but not necessarily the standard ones.

We also note that if P exists then $\nabla := \lambda x.xx$ ("self-apply") is contained in P.

Rather annoyingly, and despite considerable effort, we are currently unable to prove either the existence or the non-existence of a set P satisfying the condition above.

6.3 Non example: Hereditarily total functions

To see that not all seemingly reasonable mixed-variance equations on predicates have solutions consider the predomain D from the previous example and ask whether there exists $T \subseteq D$ (admissible or not) such that

$$f \in T \iff \forall x \in T. f(x) \neq \perp \wedge f(x) \in T$$

We think of T as comprising the "hereditarily total" functions. Obviously, we can define $T_0 \subseteq D$ by

$$f \in T_0 \iff \forall x \in D. f(x) \neq \perp$$

but then even the innocuous application functional $\lambda x.\lambda y.xy$ would not be in T_0. Note that it is in T if T exists.

Now, assume T did exist. As in the case of the challenge P above, we find $\nabla \in T$. But this would imply $\nabla\nabla \neq \perp$. However, it is well-known that $\nabla\nabla = \perp$.

7. Meaning of types

Our aim is to associate with every type A (and store typing Σ and location set Π such that $\Pi \vdash \Sigma\ ok$ and $\Pi \vdash A\ ok$) an admissible PER $[\![\Pi; \Sigma \vdash A]\!]$ on $\mathbf{V}$ and, if additionally $\Pi \vdash \varepsilon\ ok$, an admissible PER $\mathtt{T}_{[\![\Pi;\Sigma\vdash\varepsilon]\!]}([\![\Pi; \Sigma \vdash A]\!])$ on computations in such a way that evaluation of an expression in two equilavent environments gives rise to two equivalent computations. Formally,

1. if $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$ and $(\theta(x), \theta'(x)) \in [\![\Pi; \Sigma \vdash \Theta(x)]\!]$ for all $x \in \mathrm{dom}(\Theta)$ (written $(\theta,\theta') \in [\![\Pi; \Sigma \vdash \Theta]\!]$ for short) then $([\![e]\!]\theta, [\![e]\!]\theta') \in \mathtt{T}_{[\![\Pi;\Sigma\vdash\varepsilon]\!]}([\![\Pi; \Sigma \vdash A]\!])$;
2. if $\Pi; \Sigma; \Theta \vdash e_1 = e_2 : A, \varepsilon$ and $(\theta,\theta') \in [\![\Pi; \Sigma \vdash \Theta]\!]$ then $([\![e_1]\!]\theta, [\![e_2]\!]\theta') \in \mathtt{T}_\varepsilon([\![\Pi; \Sigma \vdash A]\!])$
3. If $(f,f') \in \mathtt{T}_{[\![\Pi;\Sigma\vdash\varepsilon]\!]}([\![\Pi; \Sigma \vdash \mathtt{bool}]\!])$ then $f(s_\Sigma) = \perp \iff f'(s_\Sigma) = \perp$ and if $f(s_\Sigma) = (s,b)$ and $f'(s_\Sigma) = (s',b')$ then $b = b'$.

It is well-known that these properties ensure soundness of the equational theory with respect to typed observational equivalence, see e.g. [8].

Definition 7 (specifications)**.** *A* type specification *is an* admissible *PER on* $\mathbf{V}$.

A computation specification *is an* admissible *PER on* $\mathbf{C}$ *(recall that* $\mathbf{C} = \mathbf{S} \to (\mathbf{S} \times \mathbf{V})_\perp$*) which contains the pair* $(\lambda s.\perp, \lambda s.\perp)$*, i.e., relates the constantly* $\perp$ *function to itself.*

A store relation *is a relation on stores* $\mathbf{S}$ *not necessarily admissible or a PER.*

Definition 8. *If $L \subseteq \mathbb{L}$ and $s, s_1 \in \mathbf{S}$, we write $s \sim_L s_1$ to mean that $s.\ell = s_1.\ell$ holds for all $\ell \in L$.*

An effect specification *is a set E of store relations.*

These concepts allows us to interpret type formers semantically as operations on type and effect specifications.

Proposition 9. *Let Π be a set of locations. Consider the definitions of families given in Figure 7.*

1. $\mathtt{Int}, \mathtt{Bool}, \mathtt{Unit}$ *are type specifications;*
2. *If A, B are type specifications then $\mathtt{Prod}A, B$ is a type specification;*
3. *If A is a type specification and Q is a computation specification then $\mathtt{Arr}A, Q$ is a type specification;*
4. *If E is an effect specification and A is a type specification then $\mathtt{T}_E(A)$ is a computation specification.*

We remark that, unlike in our earlier work [3], the relation $\mathtt{T}^O_E(A)$, while being admissible, is not in general transitive, hence the need to take its admissible PER-closure. The failure of transitivity is due to the fact that the identity store relation that would be used in a proof of transitivity cannot be assumed to be a member of E.

Definition 10 (Interpretation of effects)**.** *Let $\ell \in \mathbb{L}$ and A be a type specification. The effect specifications $\mathtt{Rd}_{\ell,A}, \mathtt{Wr}_{\ell,A}$ representing reading and writing of values specified by A at location ℓ are defined in Figure 8.*

$$
\begin{array}{lcl}
R \in \mathtt{Rd}_{\ell,A} & \iff & sRs' \Rightarrow s.\ell A s'.\ell \\
R \in \mathtt{Wr}_{\ell,A} & \iff & sRs' \wedge (v,v') \in A \Rightarrow s[\ell \mapsto v] R s[\ell \mapsto v']
\end{array}
$$

Figure 8. Semantic representations of reading and writing.

Definition 11. *If $L \subseteq \mathbb{L}$ and $X = (X_\ell)_{\ell \in L}$ is an L-indexed family of type specifications then we define a store relation $R_{L,X}$ by*

$$(s,s') \in R_{L,X} \iff \forall \ell \in L.(s.\ell, s'.\ell) \in X_\ell$$

If Y is another type specification and $Z = (Z_\ell)_{\ell \in L}$ is another L-indexed family of type specifications then we define a computation specification $\mathtt{T}^O_{L,X,Z}(Y)$ by

$$
\begin{array}{l}
(f,f') \in \mathtt{T}^O_{L,X,Z}(Y) \iff \\
\quad \forall s\ s'\ s_1\ s_1'\ v\ v'.sR_{L,X}s' \Rightarrow \\
\quad\quad ((f\ s = \bot \Leftrightarrow f'\ s' = \bot) \wedge \\
\quad\quad ((f\ s) = (s_1,v) \wedge (f'\ s') = (s_1',v') \Rightarrow \\
\quad\quad\quad (s_1 R_{L,Z} s_1' \wedge (v,v') \in Y) \wedge s_1 \sim_{\mathbb{L}\setminus L} s \wedge s_1' \sim_{\mathbb{L}\setminus L} s'))
\end{array}
$$

Lemma 12. *If $L \subseteq \mathbb{L}$ is finite and $X = (X_\ell)_{\ell \in L}$ is an L-indexed family of type specifications and Y is another type specification then*

$$\mathtt{T}^O_{\bigcap_{\ell \in L} \mathtt{Wr}_{\ell,X_\ell} \cap \mathtt{Rd}_{\ell,X_\ell}}(Y) = \mathtt{T}^O_{L,X,X}(Y)$$

Proof. For the "$\subseteq$" direction assume

$$(f,f') \in \mathtt{T}^O_{\bigcap_{\ell \in L} \mathtt{Wr}_{\ell,X_\ell} \cap \mathtt{Rd}_{\ell,X_\ell}}(Y)$$

and $sR_{L,X}s'$. Define R^* by

$$R^* = \{(t,t') \mid tR_{L,X}t' \wedge t \sim_{\mathbb{L}\setminus L} s \wedge t' \sim_{\mathbb{L}\setminus L} s'\}$$

Clearly, $R^* \in \bigcap_{\ell \in L} \mathtt{Wr}_{\ell,X_\ell} \cap \mathtt{Rd}_{\ell,X_\ell}$ and sR^*s'. We then get $f\ s = \bot \iff f's' = \bot$ and if $f\ s = (s_1,v)$ and $f'\ s' = (s_1',v')$ then $s_1 R^* s_1'$ and $(v,v') \in Y$. The claim follows from the definition of R^*.

For the "$\supseteq$" direction we assume $(f,f') \in \mathtt{T}^O_{L,X,X}(Y)$ and $R \in \bigcap_{\ell \in L} \mathtt{Wr}_{\ell,X_\ell} \cap \mathtt{Rd}_{\ell,X_\ell}$ and sRs'. Now $sR_{L,X}s'$ as $R \in \bigcap_{\ell \in L} \mathtt{Rd}_{\ell,X_\ell}$. Thus $f\ s = \bot \iff f'\ s' = \bot$ by assumption. If $f\ s = (s_1,v)$ and $f'\ s' = (s_1',v')$ then our assumption on f, f' yields $s_1 R_{L,X} s_1'$ and $(v,v') \in Y$.

Enumerate L as $L = \{\ell_1, \ldots, \ell_n\}$. Since $s_1 \sim_{\mathbb{L}\setminus L} s$, we have $s_1 = s[\ell_1 \mapsto s_1.\ell_1] \ldots [\ell_n \mapsto s_1.\ell_n]$ and similarly for s_1'.

From $s_1 R_{L,X} s_1'$ and $R \in \bigcap_{\ell \in L} \mathtt{Wr}_{\ell,X_\ell}$ we can then conclude $s_1 R s_1'$. □

8. Logical Relation

For each type A and store typing Σ such that $\Pi \vdash A\ ok$ and $\Pi \vdash \Sigma\ ok$ we will define a type specification $[\![\Pi; \Sigma \vdash A]\!]$ such that the clauses in Figure 9 are satisfied. The semantic meanings of

$$
\begin{array}{rcl}
[\![\Pi; \Sigma \vdash \mathtt{unit}]\!] & = & \mathtt{Unit} \\
[\![\Pi; \Sigma \vdash \mathtt{int}]\!] & = & \mathtt{Int} \\
[\![\Pi; \Sigma \vdash \mathtt{bool}]\!] & = & \mathtt{Bool} \\
[\![\Pi; \Sigma \vdash A \times B]\!] & = & \mathtt{Prod}[\![\Pi; \Sigma \vdash A]\!], [\![\Pi; \Sigma \vdash B]\!] \\
[\![\Pi; \Sigma \vdash A \xrightarrow{\varepsilon} B]\!] & = & \mathtt{Arr}[\![\Pi; \Sigma \vdash A]\!], \mathtt{T}_{[\![\Pi;\Sigma \vdash \varepsilon]\!]}([\![\Pi; \Sigma \vdash B]\!]) \\
[\![\Pi; \Sigma \vdash \emptyset]\!] & = & \text{"all store relations"} \\
[\![\Pi; \Sigma \vdash \varepsilon_1 \cup \varepsilon_2]\!] & = & [\![\Pi; \Sigma \vdash \varepsilon_1]\!] \cap [\![\Pi; \Sigma \vdash \varepsilon_2]\!] \\
[\![\Pi; \Sigma \vdash \{rd_\ell\}]\!] & = & \mathtt{Rd}_{\ell,[\![\Pi;\Sigma \vdash \Sigma(\ell)]\!]} \\
[\![\Pi; \Sigma \vdash \{wr_\ell\}]\!] & = & \mathtt{Wr}_{\ell,[\![\Pi;\Sigma \vdash \Sigma(\ell)]\!]}
\end{array}
$$

Figure 9. Logical relation

type formers used therein are those defined in Figure 7.

As the clauses in Fig. 9 cannot be read as a definition (inductive or recursive) we must prove that a relation satisfying them does indeed exist.

Theorem 2. *There is a function $[\![-]\!]$ such that whenever $\Pi \vdash A\ ok$ and $\Pi \vdash \Sigma\ ok$ then $[\![\Pi; \Sigma \vdash A]\!]$ is a type specification for Π and the equations in Figure 9 are satisfied.*

Proof. We will first define $[\![\Pi; \Sigma \vdash A]\!]$ for storable A. Once this is in place we can use the clauses in Figure 9 to extend to all types by induction on types.

In this case, by Lemma 12 the equation governing $[\![\Pi; \Sigma \vdash A \xrightarrow{\varepsilon} B]\!]$ is equivalent to

$$[\![\Pi; \Sigma \vdash A \xrightarrow{\varepsilon} B]\!] = \mathtt{Arr}[\![\Pi; \Sigma \vdash A]\!], \mathrm{per}(\mathtt{T}^O_{L,X,X}([\![\Pi; \Sigma \vdash B]\!])) \tag{1}$$

where L comprises the locations mentioned in ε and $X_\ell = [\![\Pi; \Sigma \vdash \Sigma(\ell)]\!]$.

Thus, it suffices to "solve" the equation system with this replacement made. In order to do so, we employ Theorem 1 on the following predomain D and subset $\mathcal{V}$ of $\mathrm{Adm}(D)$.

Fix $\Pi; \Sigma$ and let T be the set of storable types involving locations from Π. The predomain D then is the product

$T \times \mathbf{V} \times \mathbf{V}$. Note that $V \subset D$ is admissible if for each $A \in T$ and ascending chains $(v_i)_i, (v_i')_i$ with $(A, v_i, v_i') \in V$ for all i one has $(A, \bigsqcup_i v_i, \bigsqcup_i v_i') \in V$.

The set $\mathcal{V}$ then comprises all those admissible subsets V of D which have the property that for each $A \in T$ the set $\{(v,v') \mid (A,v,v') \in V\}$ is a(n) (admissible) PER.

Let us denote a typical element of $\mathcal{V}$ by $(\!|-|\!)$ (possibly decorated) writing $(v,v') \in (\!|A|\!)$ if $(A,v,v') \in (\!|-|\!)$.

$$
\begin{array}{rcl}
(v,v') \in \mathtt{Unit} & \iff & v = \mathit{unit}() = v' \\
(v,v') \in \mathtt{Int} & \iff & \exists x.v = \mathit{int}(x) \wedge v' = \mathit{int}(x) \\
(v,v') \in \mathtt{Bool} & \iff & \exists x.v = \mathit{bool}(x) \wedge v' = \mathit{bool}(x) \\
(v,v') \in \mathtt{Prod}A,B & \iff & \exists v_1\ v_2\ v_1'\ v_2'.v = \mathit{pair}(v_1,v_2) \wedge v' = \mathit{pair}(v_1',v_2') \wedge (v_1,v_1') \in A \wedge (v_2,v_2') \in B \\
(v,v') \in \mathtt{Arr}A,Q & \iff & \exists f\ f'.v = \mathit{fun}(f) \wedge v' = \mathit{fun}(f') \wedge \forall (a,a') \in A.(f\ a, f'\ a') \in Q \\
\mathtt{T}_E(A) & = & \mathrm{per}(\mathtt{T}_E^O(A)) \\
(f,f') \in \mathtt{T}_E^O(A) & \iff & \forall s\ s'\ s_1\ s_1'\ v\ v'.\forall R \in E.(sRs' \Rightarrow \\
& & \quad (f\ s = \bot \Leftrightarrow f'\ s' = \bot) \wedge \\
& & \quad ((f\ s) = (s_1,v) \wedge (f'\ s') = (s_1',v') \Rightarrow s_1 R s_1' \wedge (v,v') \in A)
\end{array}
$$

Figure 7. Semantics of type formers

Given $(\!|-|\!)^-, (\!|-|\!)^+ \in \mathcal{V}$, we define $(\!|-|\!) := F((\!|-|\!)^-, (\!|-|\!)^+)$ by

$$
\begin{array}{rcl}
(\!|\mathtt{unit}|\!) & = & \mathtt{Unit} \\
(\!|\mathtt{int}|\!) & = & \mathtt{Int} \\
(\!|\mathtt{bool}|\!) & = & \mathtt{Bool} \\
(\!|A \times B|\!) & = & \mathtt{Prod}(\!|A|\!)^+, (\!|B|\!)^+ \\
(\!|A \xrightarrow{\varepsilon} B|\!) & = & \mathtt{Arr}(\!|A|\!)^-, \mathrm{per}(\mathtt{T}^O_{\mathrm{locs}(\varepsilon),X^-,X^+}((\!|B|\!)^+)) \\
 & & \text{where } X_\ell^{+/-} = (\!|\Sigma(\ell)|\!)^{+/-}
\end{array}
$$

Now given a subset $(\!|-|\!)$ satisfying $(\!|-|\!) = F((\!|-|\!), (\!|-|\!))$ we can put $[\![\Pi; \Sigma \vdash A]\!] := (\!|A|\!)$ (recall that Π, Σ were temporarily fixed) and thus satisfy the equations in Fig. 9 for storable A. As already mentioned, for general A the equations can then be understood as an inductive definition.

It remains to show using Thm. 1 that such fixpoint $(\!|-|\!)$ indeed exists. We define $(\!|-|\!)_0^-$ and $(\!|-|\!)_0^+$ by

$$
\begin{array}{rcl}
(\!|A|\!)_0^- & = & \mathbf{V} \times \mathbf{V} \\
(\!|-|\!)_0^+ & = & F((\!|-|\!)_0^-, (\!|-|\!)_{-1}^+)
\end{array}
$$

where $(\!|A|\!)_{-1}^+ = \emptyset$. The required functions $q_i : D \to D$ are given by

$$q_i(A,v,v') = (A, q_i^A(v), q_i^A(v'))$$

where the functions q_i^A are defined in Fig. 10

Lemma 13. *The functions q_i are retractions and it holds that $q_i(A,v,v') \sqsupseteq (A, p_i(v), p_i(v'))$.*

Proof. Straightforward induction on i. □

Lemma 14. *1. For any $(\!|-|\!)^+, (\!|-|\!)^- \in \mathcal{V}$ one has $(\!|-|\!)_0^+ \subseteq F((\!|-|\!)^-, (\!|-|\!)^+) \subseteq (\!|-|\!)_0^-$,*
2. $q_0 : (\!|-|\!)_0^- \to (\!|-|\!)_0^+$
3. If $q_i : (\!|-|\!)^-, (\!|-|\!)^+$ then $q_{i+1} : F((\!|-|\!)^+, (\!|-|\!)^-) \to F((\!|-|\!)^-, (\!|-|\!)^+)$

Proof. The first two items are obvious. We consider the following, most interesting, subcase of the third one: suppose that $A \in T$ and L is a finite set of locations (typically $L = \mathrm{locs}(\varepsilon)$).

Suppose furthermore that $q_i : (\!|-|\!)^- \to (\!|-|\!)^+$.

Define $X_\ell^{+/-} = (\!|\Sigma(\ell)|\!)^{+/-}$. We have to show that

$$q_{i+1}^{T_L(A)} : \mathtt{T}^O_{L,X^+,X^-}((\!|A|\!)^-) \to \mathtt{T}^O_{L,X^-,X^+}((\!|A|\!)^+)$$

So suppose that

$$(f,f') \in \mathtt{T}^O_{L,X^+,X^-}((\!|A|\!)^-) \tag{2}$$

$$sR_{L,X^-}s' \tag{3}$$

$$
\begin{array}{rcl}
q_i^A & : & \mathbf{V} \to \mathbf{V} \\
q_i^{T_\varepsilon A} & : & \mathbf{C} \to \mathbf{C} \\
q_i^L & : & \mathbf{S} \to \mathbf{S}
\end{array}
$$

$$
\begin{array}{rcl}
q_i^A(x) & = & x \text{ when } A \text{ is a base type} \\
q_i^{A\times B}(\mathit{pair}(v_1,v_2)) & = & \mathit{pair}(q_i^A(v_1), q_i^B(v_2)) \\
q_i^{A\xrightarrow{\varepsilon}B}(\mathit{fun}(f)) & = & \mathit{fun}(q_i^A; f; q_i^{T_\varepsilon B}) \\
q_i^{T_\varepsilon B}(f)(s) & = & \bot \text{ when } f(q^{\mathrm{locs}(\varepsilon)}(s)) = \bot \\
q_0^{T_\varepsilon B}(f)(s) & = & \bot \\
q_{i+1}^{T_\varepsilon B}(f)(s) & = & \bot \text{ when } f(q_i^{\mathrm{locs}(\varepsilon)}(s)) = \bot \\
q_{i+1}^{T_\varepsilon B}(f)(s) & = & (q_i^{\mathrm{locs}(\varepsilon)}(s_1), q_{i+1}^B(v)) \\
 & & \text{when } f(q_i^{\mathrm{locs}(\varepsilon)}(s)) = (s_1,v) \\
q_i^L(s)(\ell) & = & q_i^{\Sigma(\ell)}(s.\ell) \text{ when } \ell \in L \\
q_i^L(s)(\ell) & = & s.\ell \text{ otherwise} \\
q_i^A(v) & = & \mathit{wrong} \text{ in all other cases}
\end{array}
$$

Figure 10. Nonstandard retractions

The assumption $q_i : (\!|-|\!)^- \to (\!|-|\!)^+$ gives us $q_i^L(s) R_{L,X^+} q_i^L(s')$.

Furthermore, we know from the definition of q^L that $q_i^L(s) \sim_{\mathbb{L}\setminus L} s$ and $q_i^L(s') \sim_{\mathbb{L}\setminus L} s'$.

Thus, $f(q_i^L(s)) = \bot \iff f'(q_i^L(s')) = \bot$. Suppose now that $f(q_i^L(s)) = (t_1, v_1)$ and $f'(q_i^L(s')) = (t_1', v_1')$. We then know that $t_1 R_{L,X^-} t_1'$ and $(v_1, v_1') \in (\!|A|\!)^-$ and $t_1 \sim_{\mathbb{L}\setminus L} s$ and $t_1' \sim_{\mathbb{L}\setminus L} s'$. We now have $q^{T_L(A)}(f)(s) = (s_1, v)$ and $q^{T_L(A)}(f)(s) = (s_1', v')$ with $s_1 = q_i^L(t_1)$ and $s_1' = q_i^L(t_1')$ and $v = q_i^A(v_1)$ and $v' = q_i^A(v_1')$. We thus get $s_1 R_{L,X^+} s_1'$ and $(v,v') \in (\!|A|\!)^+$. Finally, $s_1 \sim_{\mathbb{L}\setminus L} s$ and $s_1' \sim_{\mathbb{L}\setminus L} s'$ since q^L does not modify $\mathbb{L} \setminus L$. □

This lemma completes the premises to Theorem 1 which then yields the existence of a (unique) subset $(\!|-|\!) \in \mathcal{V}$ satisfying $F((\!|-|\!), (\!|-|\!)) = (\!|-|\!)$. We then put $[\![\Pi; \Sigma \vdash A]\!] := (\!|A|\!)$. Lemma 12 shows that this definition satisfies the equations in Fig. 9. Since $\Pi; \Sigma$ was arbitrary this defines $[\![-]\!]$ for all storable types. We then extend $[\![-]\!]$ to all types using the equations from Fig. 9 as definitions noticing that the meaning of non-storable types only

depends on the (already defined) meaning of storable types and the meaning of smaller non-storable types. This concludes the proof of Theorem 2. □

We can now formulate the following 'fundamental theorem' of logical relations, which states that terms are related to themselves. We write $(\theta, \theta') \in [\![\Pi; \Sigma \vdash \Theta]\!]$ if $\mathrm{dom}(\Theta) \subseteq \mathrm{dom}(\theta) \cap \mathrm{dom}(\theta')$, and for all $x \in \mathrm{dom}(\Theta).(\theta x, \theta' x) \in [\![\Pi; \Sigma \vdash \Theta x]\!]$.

Theorem 3 (Fundamental Theorem)**.** *If* $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$ *and* $(\theta, \theta') \in [\![\Pi; \Sigma \vdash \Theta]\!]$*, then*

$$([\![e]\!]\theta, [\![e]\!]\theta') \in T_{[\![\Pi;\Sigma\vdash\varepsilon]\!]}[\![\Pi; \Sigma \vdash A]\!].$$

Proof. By induction on the derivation of $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$. The different cases are all direct which is no surprise since the logical relation has been defined so as to make the fundamental theorem true. We only treat the cases T-REC, T-READ, T-WRITE, and T-LET here the others being essentially equivalent to the corresponding ones in [3].

Case T-REC*:* Here $e = \mathtt{rec}\, f\, x.e_0$ and $A = A_1 \xrightarrow{\varepsilon_0} A_2$ and $\varepsilon = \emptyset$ and $\Pi; \Sigma; \Theta, f{:}A_1 \xrightarrow{\varepsilon_0} A_2, x{:}A_1 \vdash e_0 : A_2$.

Recall that $[\![e]\!]\theta = \bigsqcup_i g_i$ where $g_0 = \lambda x.\lambda s.\bot$ and

$$g_{i+1} = \lambda v.[\![e_0]\!]\theta[x{\mapsto}v, f{\mapsto}\mathit{fun}(g_i)]$$

Define g'_i analogously with θ replaced by θ'.

We show by subordinate induction on i that

$$(g_i, g'_i) \in [\![\Pi; \Sigma \vdash A \xrightarrow{\varepsilon_0} B]\!]$$

holds for all i.

For $i = 0$ this is obvious from the definition. For $i > 0$ this follows from the main induction hypothesis applied to e_0 and the subordinate induction hypothesis. We then get the desired conclusion using admissibility of $[\![-]\!]$.

Case T-READ*:* Here $e = !\ell$ and $A = \Sigma(\ell)$ and $\varepsilon = \{rd_\ell\}$. Suppose that $R \in [\![\Pi; \Sigma \vdash \varepsilon]\!]$ and sRs'. We then have $s_1 = s$, $s'_1 = s'$, and $v = s.\ell$ and $v' = s'.\ell$ (with the usual notation).

Clearly, $s_1 R s'_1$ and also $(v, v') \in [\![\Sigma(\ell)]\!]$ since $R \in [\![\Pi; \Sigma \vdash \varepsilon]\!]$.

Case T-WRITE*:* Here $e = \ell := x$ and $A = \mathtt{unit}$ and $\Theta(x) = \Sigma(\ell)$ and $\varepsilon = \{wr_\ell\}$. Suppose that $R \in [\![\Pi; \Sigma \vdash \varepsilon]\!]$ and sRs'. We have $s_1 = s[\ell{\mapsto}\theta(x)]$ and $s'_1 = s'[\ell{\mapsto}\theta'(x)]$. We know $(\theta(x), \theta'(x)) \in [\![\Pi; \Sigma \vdash \Sigma(\ell)]\!]$ from the assumption on Θ and so $s_1 R s'_1$ since $R \in [\![\Pi; \Sigma \vdash \varepsilon]\!]$.

Case T-LET*:* We use Prop. 3 to "strip off" the admissible PER closure $\mathrm{per}(-)$ from the induction hypothesis and are thus lead to essentially the situation of [3], see also [8] for a similar argument. □

Definition 15. *If* $\Pi \vdash \Sigma$ *ok and* $\Pi \vdash A$ *ok and* $\Pi \vdash \varepsilon$ *ok then we write* $\Pi; \Sigma; \Theta \models e_1 = e_2 : A, \varepsilon$ *to mean that whenever* $(\theta, \theta') \in [\![\Pi; \Sigma \vdash \Theta]\!]$ *then* $([\![e_1]\!]\theta, [\![e_2]\!]\theta') \in [\![\Pi; \Sigma \vdash A]\!]$.

Theorem 4. *Whenever* $\Pi; \Sigma; \Theta \vdash e_1 = e_2 : A, \varepsilon$ *is derivable with the rules in Fig. 6 then* $\Pi; \Sigma; \Theta \models e_1 = e_2 : A, \varepsilon$.

Proof. The proof follows the same pattern as the one of the corresponding result in [3], and can be found in an Appendix available online `http://www.tcs.ifi.lmu.de/~mhofmann/papers/ppdp09.pdf`. Again, we use Prop. 3 to "strip off" $\mathrm{per}(-)$ operators in instances of the induction hypothesis.

We briefly sketch some auxiliary lemmas, as well as interesting cases of the main result. We write $[\![\varepsilon]\!]$ for $[\![\Pi; \Sigma \vdash \varepsilon]\!]$ if Π and Σ are obvious from the context, and similarly for $[\![A]\!]$ and $[\![\Theta]\!]$.

Lemma 16. *(No reads) Let* $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$*,* $(\theta, \theta') \in [\![\Theta]\!]$*,* $[\![e]\!]\theta s = (s_1, v)$ *and* $[\![e]\!]\theta' s' = (s'_1, v')$*. Furthermore, let* $(s.\ell, s'.\ell) \in [\![\Sigma(\ell)]\!]$ *hold for all* $\ell \in \mathrm{rds}(\varepsilon)$*. Then,* $(v, v') \in [\![A]\!]$*, and for all* $\ell \in \mathrm{wrs}(\varepsilon)$*,*

- $s_1.\ell = s.\ell$ *and* $s'_1.\ell = s'.\ell$*, i.e.* ℓ *is not written at all, or*
- $(s_1.\ell, s'_1.\ell) \in [\![\Sigma(\ell)]\!]$*, i.e.* ℓ *is updated with related values.*

For the proof of this lemma we define

$$R = \left\{ (t, t') \;\middle|\; \begin{array}{c} \forall\, \ell \in \mathrm{rds}(\varepsilon).\ (t.\ell, t'.\ell) \in [\![\Sigma(\ell)]\!] \wedge \\ \forall\, \ell \in \mathrm{wrs}(\varepsilon).\ (t.\ell, t'.\ell) \in [\![\Sigma(\ell)]\!] \vee \\ (t.\ell = s.\ell \wedge t'.\ell = s'.\ell) \wedge \\ \forall\, \ell \in \mathbb{L} \setminus (\mathrm{rds}(\varepsilon) \cup \mathrm{wrs}(\varepsilon)). \\ (t.\ell = s.\ell \wedge t'.\ell = s'.\ell) \end{array} \right\},$$

observe that $R \in [\![\varepsilon]\!]$ and sRs' hold, and use the fundamental theorem to obtain $(v, v') \in [\![A]\!]$ and $s_1 R s'_1$. From this, the claim follows by the definition of R.

In a similar manner, we prove the following two lemmas, where we write $s \models_\varepsilon \Sigma$ if all $\ell \in \mathrm{rds}(\varepsilon)$ satisfy $(s.\ell, s.\ell) \in [\![\Sigma(\ell)]\!]$.

Lemma 17. *(No writes) Let* $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$*,* $(\theta, \theta') \in [\![\Theta]\!]$*,* $[\![e]\!]\theta s = (s_1, v)$*, and* $s \models_\varepsilon \Sigma$*. Then* $(v, v) \in [\![A]\!]$*,* $s_1 \models_\varepsilon \Sigma$*, and* $s \sim_{\mathbb{L}\setminus\mathrm{wrs}(\varepsilon)} s_1$*.*

Lemma 18. *(Termination) Let* $\Pi; \Sigma; \Theta \vdash e : A, \varepsilon$*,* $(\theta, \theta') \in [\![\Theta]\!]$*,* $s \models_\varepsilon \Sigma$*, and* $s \sim_{\mathrm{rds}(\varepsilon)} s'$*. Then* $[\![e]\!]\theta s = \bot \iff [\![e]\!]\theta' s' = \bot$*.*

The main result is then shown by induction on the rules from Figure 6. We treat one representative case.

Rule E-DUP . We have $e_1 \equiv \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ \mathit{pair}(x, x)$ and $e_2 \equiv \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow e\ \mathtt{in}\ \mathit{pair}(x, y)$ for some $\Pi; \Sigma; \Theta \vdash e : A', \varepsilon$ with $x \notin \mathrm{dom}(\Theta)$, $A \equiv A' \times A'$ and $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset$. For $(\theta, \theta') \in [\![\Theta]\!]$ we have to show

$$(f, f') := ([\![e_1]\!]\theta, [\![e_2]\!]\theta') \in \mathtt{T}_{[\![\varepsilon]\!]}([\![A]\!]) = \mathrm{per}(\mathtt{T}^O_{[\![\varepsilon]\!]}([\![A]\!])).$$

By the definition of the operator $\mathrm{per}(.)$, it suffices to show $(f, f') \in \mathtt{T}^O_{[\![\varepsilon]\!]}([\![A]\!])$, so let $R \in [\![\varepsilon]\!]$ and sRs'.

The fundamental theorem applied to $(\theta, \theta') \in [\![\Theta]\!]$ and $\Pi; \Sigma; \Theta \vdash e : A', \varepsilon$ yields $([\![e]\!]\theta, [\![e]\!]\theta') \in \mathtt{T}_{[\![\varepsilon]\!]}([\![A']\!]) = \mathrm{per}(\mathtt{T}^O_{[\![\varepsilon]\!]}([\![A']\!]))$. Using Proposition 3 we deduce $([\![e]\!]\theta, [\![e]\!]\theta') \in \mathtt{T}^O_{[\![\varepsilon]\!]}([\![A']\!])$, so from $R \in [\![\varepsilon]\!]$ and sRs' we deduce that $[\![e]\!]\theta s = \bot$ is equivalent to $[\![e]\!]\theta' s' = \bot$. Thus, if $[\![e]\!]\theta s = \bot$ then $f' s' = \bot$ follows.

So let $[\![e]\!]\theta s = (s_1, u)$ and $[\![e]\!]\theta' s' = (s'_1, u')$. By the fundamental theorem, we have

i) $s_1 R s'_1$ and

ii) $(u, u') \in [\![A']\!]$.

We also observe that

iii) $(\theta', \theta') \in [\![\Theta]\!]$ holds, by using $(\theta, \theta') \in [\![\Theta]\!]$ and (pointwise) per-ness of $[\![\Theta]\!]$

iv) $s' \models_\varepsilon \Sigma$, since for all $\ell \in \mathrm{rds}(\varepsilon)$, sRs' implies $(s.\ell, s'.\ell) \in [\![\Sigma(\ell)]\!]$, so by per-ness of $[\![\Sigma(\ell)]\!]$ we have $(s'.\ell, s'.\ell) \in [\![\Sigma(\ell)]\!]$

We can thus apply Lemma 17 to $[\![e]\!]\theta' s' = (s'_1, u')$, and obtain

v) $(u', u') \in [\![A']\!]$, and

vi) $s'_1 \models_\varepsilon \Sigma$, and

vii) $s' \sim_{\mathbb{L}\setminus\mathrm{wrs}(\varepsilon)} s'_1$.

From $s' \sim_{\mathbb{L}\setminus\mathrm{wrs}(\varepsilon)} s'_1$ and $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset$ we obtain

viii) $s' \sim_{\mathrm{rds}(\varepsilon)} s'_1$, and

ix) $(\theta', \theta'[x \mapsto u']) \in [\![\Theta]\!]$ holds due to $(\theta', \theta') \in [\![\Theta]\!]$ and $x \notin \mathrm{dom}(\Theta)$

so we can first apply Lemma 18 and obtain $[\![e]\!]\theta'[x \mapsto u']s'_1 = (s'_2, v')$ for some s'_2 and v'. We thus have

$$fs = (s_1, (u, u)) \text{ and } f's' = (s'_2, (u', v')),$$

hence all that remains to show is $s_1 R s'_2$ and $(u, v') \in [\![A']\!]$. Indeed, we may apply Lemma 16 to $[\![e]\!]\theta s = (s_1, u)$ and $[\![e]\!]\theta'[x \mapsto u']s'_1 = (s'_2, v')$, since $(\theta, \theta'[x \mapsto u']) \in [\![\Theta]\!]$ (from $(\theta, \theta') \in [\![\Theta]\!]$ and ix)) and all $\ell \in \mathrm{rds}(\varepsilon)$ satisfy $(s.\ell, s'.\ell) \in [\![\Sigma(\ell)]\!]$ (due to sRs') and $s'.\ell = s'_1.\ell$ (due to $viii$)), hence $(s.\ell, s'_1.\ell) \in [\![\Sigma(\ell)]\!]$. We thus obtain the desired $(u, v') \in [\![A']\!]$, and that

x) all $\ell \in \mathrm{wrs}(\varepsilon)$ satisfy $s_1.\ell = s.\ell \wedge s'_2.\ell = s'_1.\ell$ or $(s_1.\ell, s'_2.\ell) \in [\![\Sigma(\ell)]\!]$.

We may also apply Lemma 16 to $[\![e]\!]\theta s = (s_1, u)$ and $[\![e]\!]\theta' s' = (s'_1, u')$, since $(\theta, \theta') \in [\![\Theta]\!]$) and all $\ell \in \mathrm{rds}(\varepsilon)$ satisfy $(s.\ell, s'.\ell) \in [\![\Sigma(\ell)]\!]$ (due to sRs'). We thus obtain $(u, u') \in [\![A']\!]$ (again) and that

xi) all $\ell \in \mathrm{wrs}(\varepsilon)$ satisfy $s_1.\ell = s.\ell \wedge s'_1.\ell = s'.\ell$ or $(s_1.\ell, s'_1.\ell) \in [\![\Sigma(\ell)]\!]$.

Combining x) and xi) yields that each $\ell \in \mathrm{wrs}(\varepsilon)$ satisfies one of the three cases

- $(s_1.\ell, s'_2.\ell) \in [\![\Sigma(\ell)]\!]$,
- $s_1.\ell = s.\ell \wedge s'_2.\ell = s'_1.\ell$ and $s_1.\ell = s.\ell \wedge s'_1.\ell = s'.\ell$, hence $s_1.\ell = s.\ell \wedge s'_2.\ell = s'.\ell$,
- $s_1.\ell = s.\ell \wedge s'_2.\ell = s'_1.\ell$ and $(s_1.\ell, s'_1.\ell) \in [\![\Sigma(\ell)]\!]$, hence $(s_1.\ell, s'_2.\ell) = (s_1.\ell, s'_1.\ell) \in [\![\Sigma(\ell)]\!]$.

Thus, all $\ell \in \mathrm{wrs}(\varepsilon)$ satisfy $(s_1.\ell, s'_2.\ell) \in [\![\Sigma(\ell)]\!]$ or $s_1.\ell = s.\ell \wedge s'_2.\ell = s'.\ell$. For $\ell \in \mathbb{L} \setminus \mathrm{wrs}(\varepsilon)$, we have (by Lemma 17) $s_1.\ell = s.\ell \wedge s'_2.\ell = s'_1.\ell = s'.\ell$.

Consequently, we way write s_1 and s'_2 as

$$\begin{aligned} s_1 &= s[\ell_1 \mapsto v_1] \ldots [\ell_n \mapsto v_n] \\ s'_2 &= s'[\ell_1 \mapsto v'_1] \ldots [\ell_n \mapsto v'_n] \end{aligned}$$

for some (finitely many) $\ell_1, \ldots, \ell_n$ and $(v_i, v'_i) \in [\![\Sigma(\ell)]\!]$. Since R preserves write effects we conclude $s_1 R s'_2$ from sRs'.

This concludes the proof sketch of Theorem 4. □

We close with the following Lemma about relatedness of initial stores which allows us to deduce observational equivalence from semantic equality

Lemma 19. *If* $\Pi \vdash \Sigma$ *ok and* $\Pi \vdash \varepsilon$ *ok and* $R \in [\![\varepsilon]\!]$ *then* $s_\Sigma R s_\Sigma$.

9. Discussion

In this paper, we reported progress towards providing semantic justification of program transformations for effectful higher-order languages using relational techniques. Restricting our attention to a language with global store, we highlighted the need for further work on the solution theory for mixed-variance equations. Despite our inability to provide such a solution theory in its full generality at this moment in time, we believe the present paper represents substantial progress on this topic, exemplified by our ability to justify the existence of the logical relation for the language considered, which does not appear possible using existing techniques. The corresponding proof makes critical use of non-standard retractions, a technique that may indeed be a component of a general solution theory. While slight relaxations of the storability condition might be possible, new insight is required to fully eliminate it, which would be particularly welcome given the restrictions it imposes on practical programs. One would hope that once such a solution theory is in place, it would also allow one to reintegrate allocation, by combining the presented work with parameters and Kripke-interpretations of function types as demonstrated in [4]. At present it would appear possible to write down a corresponding logical relation and use it to justify transformations that involve allocation, but the precise meaning (and well-definedness) of the obtained equational theory would be unclear. In addition to this, future work will also seek to develop a better understanding of the relationship between the relationally denotational style of reasoning explored in our work, the step-indexed operational interpretations of types [1, 2], the operational theories for reasoning about observational equivalence based on bisimulation [9, 14, 13], and the approximative and inequational reasoning in the style of [5].

References

[1] A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 69–83. Springer, 2006.

[2] A. J. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), Proceedings*. IEEE Computer Society, 2002.

[3] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Reading, writing, and relations: Towards extensional semantics for effect analyses. In *4th Asian Symposium on Programming Languages and Systems (APLAS)*, LNCS, 2006.

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

[5] L. Birkedal, K. Stovring, and J. Thamsborg. Relational parametricity for references and recursive types. In *Types in Language Design and Implementation (TLDI)*, 2009.

[6] N. Bohr and L. Birkedal. Relational reasoning for recursive types and references. In Naoki Kobayashi, editor, *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Proceedings*, volume 4279 of *LNCS*, pages 79–96. Springer, 2006.

[7] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 66–77, New York, NY, USA, 2007. ACM.

[8] M. Hofmann. Correctness of effect-based program transformations. In Orna Grumberg and Tobias Nipkow, editors, *Formal Logical Methods for System Security and Correctness*, pages 149–173. IOS Press, 2008.

[9] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, pages 141–152, 2006.

[10] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127(2), 1996.

[11] B. Reus and J. Schwinghammer. Denotational semantics for Abadi and Leino's logic of objects. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings*, volume 3444 of *LNCS*, pages 263–278. Springer, 2005.

[12] J. C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1974.

[13] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), Proceedings*, pages 293–302. IEEE Computer Society, 2007.

[14] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2005.