Relational Semantics for Effect-Based Program Transformations with Dynamic Allocation

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

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

Abstract

We give a denotational semantics to a region-based effect system tracking reading, writing and allocation in a higher-order language with dynamically allocated integer references.

Effects are interpreted in terms of the preservation of certain binary relations on the store, parameterized by region-indexed partial bijections on locations.

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

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

General Terms Languages, Theory

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

1. Introduction

Many analyses and logics for imperative programs are concerned with establishing whether particular mutable variables may be read or written by a phrase. For example, the equivalence of while-programs

```
C ; if B then C' else C''  =
if B then (C;C') else (C;C'')
```

is valid when `B` does not read any variable which `C` might write. Hoare-style programming logics often have rules with side conditions on possibly-read and possibly-written variable sets, and reasoning about concurrent processes is dramatically simplified if one can establish that none of them may write a variable which another may read.

Effect systems [13, 17] are static analyses that compute upper bounds on the possible side-effects of computations. The literature contains many effect systems that analyse which storage cells may be read and which storage cells may be written (as well as many other properties), but few satisfactory accounts of the semantics of this information, or of the uses to which it may be put. Note that because effect systems *over*-estimate the possible side-effects of expressions, the information they capture is of the form that particular variables will definitely *not* be read or will definitely *not* be written. But what does that mean?

Thinking operationally, it may seem entirely obvious what is meant by saying that a variable X will not be read (written) by a command C, viz. no execution trace of C contains a read (resp. write) operation to X. But, as we have argued before [4, 7], such intensional interpretations of program properties are over-restrictive, cannot be interpreted in a standard semantics, do not behave well with respect to program equivalence or contextual reasoning and are hard to maintain during transformations. Thus we seek extensional properties that are more liberal than the intensional ones yet still validate the transformations or reasoning principles we wish to apply.

In a previous paper [8] we successfully solved this problem for a simple imperative language with global references. The main idea of that work was that a computation M preserves a store relation R provided this relation is preserved by all the side-effecting operations that M may possibly invoke. In particular, the operation "read from X" preserves all store relations R with the property that sRs' implies $s(X) = s'(X)$. The operation "write to Y", on the other hand, preserves all store relations R with the property that sRs' implies $s[Y{:=}n]Rs'[Y{:=}n]$ for all $n \in \mathbb{Z}$. Thus, if an effect system ascribes *only* the effects "read from X" and "write to Y" to a computation M then M should preserve all the relations that enjoy both of those properties. This idea extends to a compositional semantics which validates a list of effect-based program transformations, among them lifting semantically pure computations out of functions, and eliminating dead code.

In the present work we substantially extend this idea by allowing dynamic allocation of references. Accordingly, effects can no longer refer to explicit references ("read from X") but abstract from individual references using the concept of *regions* as first introduced by Lucassen and Gifford [17]. A major motivation in that work was in parallelizing compilation, though such optimizations were not presented, still less formalized. In our work we validate program equivalences for sequential programs that depend on fine-grained effect information, as in the rule for commuting computations (in C-style notation):

```
x=m1();y=m2();m3(x,y); ≡
   y=m2();x=m1();m3(x,y);
```

which requires that `m1` does not read from or write to a region that `m2` writes to and vice versa. Notice that the values `x`, `y` that are computed in different order may themselves be functions possibly embodying (freshly allocated) references.

PPDP'07 July 14–16, 2007, Wroclaw, Poland.

Notice that the equivalences we prove are generic, in that they hold for all terms with a particular effect-annotated type. Many interesting concrete instances can be shown using existing techniques, but devising schematic equational principles, driven by types rather than terms, is more challenging.

Another important addition to our earlier work is the inclusion of Lucassen-Gifford's "masking rule" which under certain circumstances allows one to remove manifestly present effect information. Effect masking was used later by Tofte and Talpin in region-based memory management [28] in order to statically infer sound deallocations. Here, we are not interested in such intensional properties of programs; rather, the masking rule merely hides non-observable side-effects, validating more program equivalences than would otherwise hold.

Our approach to the soundness of the effect analysis and to the correctness of effect-dependent program equivalences is to interpret program properties (which may be expressed as points in an abstract domain, or as non-standard types) as binary relations over a standard, non-instrumented (operational or denotational) semantics of the language. Unlike in our earlier work on global store these binary relations are now parameterized over a partially-ordered set of region layouts ("parameters"); a Kripke-style quantification over parameters is used in the definition of the relation for function types.

We stress that the contribution of this paper is not merely methodological; as far as we are aware the program equivalences in Section 7 are genuinely new results. Note that these results are phrased in terms of observational equivalences and effect typing, thus do not refer in their statement to concepts introduced here. Of course, the relational semantics which is the main technical contribution plays a crucial role in the proofs of these statements.

Acknowledgements: Support by MS Research Cambridge, EU grant EmBounded IST-FET 510255 (Hofmann), EU grant MOBIUS IST-FET-15905 (Beringer, Hofmann) is gratefully acknowledged.

2. Base Language

We study a monadically-typed, normalizing, call-by-value lambda calculus with dynamically allocated integer references. We thus extend the language from [8] but do not include recursion, references to types other than integers, etc. The use of monadic types, making an explicit distinction between values and computations, simplifies the presentation of the effect system and cleans up the equational theory of the language. A more conventionally-typed impure calculus may be translated into the monadic one via the usual 'call-by-value translation' [6], and this extends to the usual style of presenting effect systems in which every judgement has an effect, and function arrows are annotated with 'latent effects' [29].

We define value types (ranged over by A and B), computation types TA and contexts Γ as follows:

$$\begin{array}{rcl} A, B & := & \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{ref} \mid A \times B \mid A \to TB \\ \Gamma & := & x_1 : A_1, \ldots, x_n : A_n \end{array}$$

Variables are always given value types, as this is all we shall need to interpret a CBV language. There are two forms of typing judgement: value judgements $\Gamma \vdash V : A$ and computation judgements $\Gamma \vdash M : TA$, defined inductively by the rules in Figure 1. Note that the types on λ-bound variables make typing derivations unique and that addition and comparison are just representative primitive operations. To save space we omit grammars for values V and computations M as they can be inferred from the typing rules.

3. Denotational semantics

Since our simple language has no recursion, we can give it an elementary denotational semantics in the category of sets and functions.

We axiomatise states as follows, assuming a set $\mathbb{L}$ of locations and a set $\mathbb{S}$ of states. There is a constant $\emptyset \in \mathbb{S}$, the empty state. If $s \in \mathbb{S}$ then $\mathrm{dom}(s) \subseteq \mathbb{L}$ and if $\ell \in \mathrm{dom}(s)$ then $s.\ell \in \mathbb{Z}$ is a value; if $v \in \mathbb{Z}, \ell \in \mathrm{dom}(s)$ then $s[\ell \mapsto v] \in \mathbb{S}$; finally $new(s, v)$ yields a pair (ℓ, s') where $\ell \in \mathbb{L}$ and $s' \in \mathbb{S}$. These operations are subject to the following axioms:

$$\begin{array}{l} \mathrm{dom}(\emptyset) = \emptyset \\ \mathrm{dom}(s[\ell \mapsto v]) = \mathrm{dom}(s) \\ (s[\ell \mapsto v]).\ell' = \textit{if } \ell = \ell' \textit{ then } v \textit{ else } s.\ell' \\ new(s, v) = (\ell, s') \Rightarrow \mathrm{dom}(s') = \mathrm{dom}(s) \cup \{\ell\} \wedge \\ \qquad \ell \notin \mathrm{dom}(s) \wedge s'.\ell = v \end{array}$$

This abstract datatype can be implemented in a number of ways, e.g., as finite maps.[1]

If L is a set of locations, typically finite, and $s, s' \in \mathbb{S}$ then we define

$$s \sim_L s' \Leftrightarrow \mathrm{dom}(s) \supseteq L \wedge \mathrm{dom}(s') \supseteq L \wedge \forall \ell \in L. s.\ell = s'.\ell$$

The semantics of types is now given as follows:

$$\begin{array}{c} [\![\mathtt{unit}]\!] = \{\star\} \qquad [\![\mathtt{int}]\!] = \mathbb{Z} \qquad [\![\mathtt{bool}]\!] = \mathbb{B} \\ [\![\mathtt{ref}]\!] = \mathbb{L} \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!] \\ [\![A \to TB]\!] = [\![A]\!] \to [\![TB]\!] \qquad [\![TA]\!] = \mathbb{S} \to \mathbb{S} \times [\![A]\!] \end{array}$$

The interpretation of the computation type constructor is the usual state monad. The meaning of contexts is given by $[\![\Gamma]\!] = \Pi_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]$, and we can then give the semantics of judgements

$$\begin{array}{rc} & [\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!] \\ \text{and} & [\![\Gamma \vdash M : TA]\!] : [\![\Gamma]\!] \to [\![TA]\!] \end{array}$$

inductively, though we omit the completely standard details.

Figure 2 defines sequential composition as syntactic sugar and contains a few example programs with their types. The first, V_{sum} computes the sum of its three arguments using a reference to store an intermediate result. Here $\mathtt{int}^3$ abbreviates $\mathtt{int} \times (\mathtt{int} \times \mathtt{int})$ and we use an obvious pattern-matching notation to avoid accumulation of projections. The second example generates a counter object initialised at zero. Note that each evaluation of M_{cnt} produces a different counter. The next, M_{mem} is a memoised version of the successor function. It should be clear how to extend this example to a memo-functional that can memoise an argument function. The last example, M_{buf} is a function that returns the previous argument it has been called with, or zero on first invocation.

4. Effect system

We now present our effect analysis as a type system that refines the simple type system by annotating the computation type constructor with information about whether a computation may read, write, or allocate within a region.

Formally, we assume a (possibly infinite) set *Regs* of region identifiers (regions for short) ranged over by r. Types and typing judgements will involve only finitely many regions. Such finite sets are ranged over by Π. Primitive effects are al_{r} (allocation in region r), wr_{r} (write access to region r), and rd_{r} (read access to region r). An effect, ranged over by ε, is a finite set of primitive effects. If ε is an effect, we define the *reads of* ε by $\mathrm{rds}(\varepsilon) = \{\mathsf{r} \mid rd_{\mathsf{r}} \in \varepsilon\}$ and similarly its *writes* $\mathrm{wrs}(\varepsilon)$ and its *allocations* $\mathrm{als}(\varepsilon)$.

[1] This is not quite the free implementation because of the new operation.

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash b : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 + V_2 : \mathtt{int}} \qquad \frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 > V_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : B}{\Gamma \vdash (V_1, V_2) : A \times B} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i\, V : A_i}$$

$$\frac{\Gamma, x : A \vdash M : TB}{\Gamma \vdash \lambda x : A.M : A \to TB} \qquad \frac{\Gamma \vdash V_1 : A \to TB \quad \Gamma \vdash V_2 : A}{\Gamma \vdash V_1\, V_2 : TB} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{val}\, V : TA}$$

$$\frac{\Gamma \vdash M : TA \quad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N : TB} \qquad \frac{\Gamma \vdash V : \mathtt{bool} \quad \Gamma \vdash M : TA \quad \Gamma \vdash N : TA}{\Gamma \vdash \mathtt{if}\, V\, \mathtt{then}\, M\, \mathtt{else}\, N : TA}$$

$$\frac{\Gamma \vdash V : \mathtt{ref}}{\Gamma \vdash \mathtt{read}(V) : T\mathtt{int}} \qquad \frac{\Gamma \vdash V_1 : \mathtt{ref} \quad V_2 : \mathtt{int}}{\Gamma \vdash \mathtt{write}(V_1, V_2) : T\mathtt{unit}} \qquad \frac{\Gamma \vdash V : \mathtt{int}}{\Gamma \vdash \mathtt{ref}(V) : T\mathtt{ref}}$$

Figure 1. Simple computation type system

$$\begin{aligned}
M_1; M_2 &:= \mathtt{let}\ _ \Leftarrow M_1\ \mathtt{in}\ M_2 \quad \text{(sequential composition)}\\
V_{\mathrm{sum}} &:= \lambda(x,y,z){:}\mathtt{int}^3.\mathtt{let}\ x \Leftarrow \mathtt{ref}(x+y)\ \mathtt{in}\ \mathtt{read}(x) + z : \mathtt{int}^3 \to T\mathtt{int}\\
M_{\mathrm{cnt}} &:= \mathtt{let}\ x \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ (\lambda_ : \mathtt{unit}.\mathtt{read}(x), \lambda_ : \mathtt{unit}.\mathtt{write}(x, \mathtt{read}(x)+1)) :\\
&\qquad T((\mathtt{unit} \to T\mathtt{int}) \times (\mathtt{unit} \to T\mathtt{unit}))\\
M_{\mathrm{mem}} &:= \mathtt{let}\ x \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow \mathtt{ref}(1)\ \mathtt{in}\\
&\qquad \lambda a{:}\mathtt{int}.\mathtt{if}\ a = \mathtt{read}(x)\ \mathtt{then}\ \mathtt{read}(y)\ \mathtt{else}\ \mathtt{write}(x,a); \mathtt{write}(y, a+1); \mathtt{read}(y) : T(\mathtt{int} \to T\mathtt{int})\\
M_{\mathrm{buf}} &:= \mathtt{let}\ x \Leftarrow \mathtt{ref}(0)\ \mathtt{in}\ \lambda a{:}\mathtt{int}.\mathtt{let}\ o \Leftarrow \mathtt{read}(x)\ \mathtt{in}\ \mathtt{write}(x,a); o : T(\mathtt{int} \to T\mathtt{int})
\end{aligned}$$

Figure 2. Example programs

We define *refined* value types ranged over by X and Y:

$$X, Y := \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{ref}_{\mathsf{r}} \mid X \times Y \mid X \to T_\varepsilon\, Y$$

A refined computation type takes the form $T_\varepsilon\, X$ with X a refined value type. A refined typing context Θ is a finite map from variables to refined value types. The well-formedness judgements $\Pi \vdash \varepsilon\ ok$, $\Pi \vdash X\ ok$ and $\Pi \vdash \Theta\ ok$ mean that only regions declared in Π appear in ε, X and Θ. There is a subtyping relation on refined types, axiomatised in Figure 3. The erasure map, $U(\cdot)$, takes refined types to simple types (and contexts) by forgetting the effect annotations. We omit its obvious definition.

The refined type assignment system is shown in Figure 4. Note that the subject terms are the same (we still only have simple types on λ-bound variables).

Lemma 1. *If $\Pi; \Theta \vdash V : X$ then $U(\Theta) \vdash V : U(X)$, and similarly for computations.*

The last rule in Fig. 4 is called the *masking rule* which allows one to remove all reference to a region r in the effect of a computation provided this region is mentioned neither in the result type X of that computation nor in the type of any of its free variables. In the original work on region-based memory management [28] such regions could be safely deallocated upon completion of that computation. We do not consider deallocation in our system, and indeed the refined type system does not affect the semantics of computations in any way. Instead we use the masking rule to enhance applicability of effect-based program equivalences.

Note that any term typable in the original language can be typed in the refined system using a single region r and all effects set, i.e., $\Pi = \{\mathsf{r}\}$ and $\varepsilon = \{rd_{\mathsf{r}}, wr_{\mathsf{r}}, al_{\mathsf{r}}\}$ throughout. Of course, in order to maximise applicability of program equivalences and of the masking rule it is in the interest of the programmer or of an automatic type inference (which we do not consider here) to seek refined typings that use more than one region.

4.1 Example programs

Recall the example programs from Figure 2. These can be typed as follows.

$$\begin{aligned}
V_{\mathrm{sum}} &: \mathtt{int}^3 \to T_{\{al_{\mathsf{r}}, rd_{\mathsf{r}}\}}\, \mathtt{int}\\
M_{\mathrm{cnt}} &: T_{\{al_{\mathsf{r}}\}}\, ((\mathtt{unit} \to T_{\{rd_{\mathsf{r}}\}}\, \mathtt{int}) \times (\mathtt{unit} \to T_{\{wr_{\mathsf{r}}\}}\, \mathtt{unit}))\\
M_{\mathrm{mem}} &: T_{\{al_{\mathsf{r}}\}}\, (\mathtt{int} \to T_{\{wr_{\mathsf{r}}, rd_{\mathsf{r}}\}}\, \mathtt{int})\\
M_{\mathrm{buf}} &: T_{\{al_{\mathsf{r}}\}}\, (\mathtt{int} \to T_{\{wr_{\mathsf{r}}, rd_{\mathsf{r}}\}}\, \mathtt{int})
\end{aligned}$$

The function V_{sum} can even be typed as

$$V_{\mathrm{sum}} \quad : \quad \mathtt{int}^3 \to T_{\emptyset}\, \mathtt{int}$$

Thus will be (correctly) regarded as pure by our analysis. Unfortunately, the masking rule does not apply to the type of the pure computation M_{mem}. Indeed, if it did, the impure M_{buf} which has the same type as M_{mem} would be falsely given a pure typing.

$$\frac{}{X \leq X} \qquad \frac{X \leq Y \quad Y \leq Z}{X \leq Z} \qquad \frac{X \leq X' \quad Y \leq Y'}{X \times Y \leq X' \times Y'}$$

$$\frac{X' \leq X \quad T_\varepsilon\, Y \leq T_{\varepsilon'}\, Y'}{(X \to T_\varepsilon\, Y) \leq (X' \to T_{\varepsilon'}\, Y')} \qquad \frac{\varepsilon \subseteq \varepsilon' \quad X \leq X'}{T_\varepsilon\, X \leq T_{\varepsilon'}\, X'}$$

Figure 3. Subtyping refined types

$$\frac{}{\Pi;\Theta \vdash n : \mathtt{int}} \qquad \frac{}{\Pi;\Theta \vdash b : \mathtt{bool}} \qquad \frac{}{\Pi;\Theta \vdash () : \mathtt{unit}} \qquad \frac{\Pi \vdash X\ ok}{\Pi;\Theta, x{:}X \vdash x : X}$$

$$\frac{\Pi;\Theta \vdash V_1 : \mathtt{int} \quad \Pi;\Theta \vdash V_2 : \mathtt{int}}{\Pi;\Theta \vdash V_1 + V_2 : \mathtt{int}} \qquad \frac{\Pi;\Theta \vdash V_1 : \mathtt{int} \quad \Pi;\Theta \vdash V_2 : \mathtt{int}}{\Pi;\Theta \vdash V_1 > V_2 : \mathtt{bool}}$$

$$\frac{\Pi;\Theta \vdash V_1 : X \quad \Pi;\Theta \vdash V_2 : Y}{\Pi;\Theta \vdash (V_1, V_2) : X \times Y} \qquad \frac{\Pi;\Theta \vdash V : X_1 \times X_2}{\Pi;\Theta \vdash \pi_i\, V : X_i} \qquad \frac{\Pi;\Theta, x : X \vdash M : T_\varepsilon\, Y}{\Pi;\Theta \vdash \lambda x : U(X).M : X \to T_\varepsilon\, Y}$$

$$\frac{\Pi;\Theta \vdash V_1 : X \to T_\varepsilon\, Y \quad \Pi;\Theta \vdash V_2 : X}{\Pi;\Theta \vdash V_1\, V_2 : T_\varepsilon\, Y} \qquad \frac{\Pi;\Theta \vdash V : X}{\Pi;\Theta \vdash \mathtt{val}\ V : T_\emptyset\, X}$$

$$\frac{\Pi;\Theta \vdash M : T_\varepsilon\, X \quad \Pi;\Theta, x : X \vdash N : T_{\varepsilon'}\, Y}{\Pi;\Theta \vdash \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N : T_{\varepsilon \cup \varepsilon'}\, Y} \qquad \frac{\Pi;\Theta \vdash V : \mathtt{bool} \quad \Pi;\Theta \vdash M : T_\varepsilon\, X \quad \Pi;\Theta \vdash N : T_\varepsilon\, X}{\Pi;\Theta \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : T_\varepsilon\, X}$$

$$\frac{\Pi;\Theta \vdash V : \mathtt{ref}_\mathsf{r}}{\Pi;\Theta \vdash \mathtt{read}(V) : T_{\{rd_\mathsf{r}\}}\,(\mathtt{int})} \qquad \frac{\Pi;\Theta \vdash V_1 : \mathtt{ref}_\mathsf{r} \quad \Pi;\Theta \vdash V_2 : \mathtt{int}}{\Pi;\Theta \vdash \mathtt{write}(V_1, V_2) : T_{\{wr_\mathsf{r}\}}\,(\mathtt{unit})} \qquad \frac{\Pi;\Theta \vdash V : \mathtt{int} \quad \mathsf{r} \in \Pi}{\Pi;\Theta \vdash \mathtt{ref}(V) : T_{\{al_\mathsf{r}\}}\,(\mathtt{ref}_\mathsf{r})}$$

$$\frac{\Pi;\Theta \vdash V : X \quad X \leq X'}{\Pi;\Theta \vdash V : X'} \qquad \frac{\Pi;\Theta \vdash M : T_\varepsilon\, X \quad T_\varepsilon\, X \leq T_{\varepsilon'}\, X'}{\Pi;\Theta \vdash M : T_{\varepsilon'}\, X'} \qquad \frac{\Pi, \mathsf{r};\Theta \vdash M : T_\varepsilon\, X \quad \Pi \vdash X\ ok \quad \Pi \vdash \Theta\ ok}{\Pi;\Theta \vdash M : T_{\varepsilon \setminus \{rd_\mathsf{r}, wr_\mathsf{r}, al_\mathsf{r}\}}\, X}$$

Figure 4. Refined type system

A schematic example for the use of the masking rule in the spirit of Haskell's "runST" [10] is as follows. Suppose that M is a computation of an end result of type `int` employing on its way a number of counters, i.e., invocations of M_{cnt} (and possibly other stateful operations). We then get a type $T_\varepsilon\ \mathtt{int}$ for M which can then be improved to $T_\emptyset\ \mathtt{int}$ assuming that no reference is made to externally visible portions of the store (formally: the regions mentioned in ε are disjoint from regions mentioned in the types of free variables in M).

All these typings only use a single region which, however, is arbitrary and can thus be chosen differently from regions used in the context. For example, we have

$$\mathtt{let}\ x_1 \Leftarrow M_{\mathrm{buf}}\ \mathtt{in}\ \mathtt{let}\ x_2 \Leftarrow M_{\mathrm{buf}}\ \mathtt{in}\ (x_1, x_2) : T_{\{al_{\mathsf{r}_1}, al_{\mathsf{r}_2}\}}\,(J(\mathsf{r}_1) \times J(\mathsf{r}_2))$$

where $J(\mathsf{r}) = \mathtt{int} \to T_{\{wr_\mathsf{r}, rd_\mathsf{r}\}}\ \mathtt{int}$.

Remark on polymorphism It would be natural to assign a region-polymorphic type such as $\forall \mathsf{r}.T_{\{al_\mathsf{r}\}}\ J(\mathsf{r})$ to M_{buf}. In this paper we refrain from considering polymorphism (at the level of regions, effects or types) for the sake of simplicity. Our proof of type soundness can be extended to the effect-polymorphic case quite straightforwardly by interpreting polymorphic types as (possibly infinite) families of monomorphic types, as in [5].

5. Formal preliminaries

In our previous work [8] we approximated contextual equivalence by a partial equivalence relation $[\![X]\!]$ for each refined type X. In the presence of dynamic allocation such a simple-minded setup will no longer work and we move to a Kripke logical relation indexed by store layouts which we refer to as *parameters*. Parameters introduce (a) a 'representation independence' for state, capturing the fact that behaviour is invariant under permutation of locations; and (b) a distinction between observable and non-observable locations, as expressed syntactically by the masking rule.

5.1 Partial bijections

Aspect (a) of parameters is expressed by assigning to each region identifier a partial bijection between locations in the store.

Definition 1 (partial bijection)**.** *A* partial bijection *is a triple* (L, L', f) *where* L, L' *are finite subsets of* $\mathbb{L}$ *and* $f \subseteq L \times L'$ *such that* $(l_1, l_1') \in f$ *and* $(l_2, l_2') \in f$ *imply* $l_1 = l_2 \Leftrightarrow l_1' = l_2'$.

If $t = (L, L', f)$ is a partial bijection, we write $\mathrm{dom}(t) = L, \mathrm{dom}'(t) = L'$ and refer to f simply by t itself. We let (ℓ, ℓ') denote the partial bijection $(\{\ell\}, \{\ell'\}, \{(\ell, \ell')\})$, and let $\emptyset$ denote the empty partial bijection.

Two partial bijections t_1, t_2 are *disjoint* if $\mathrm{dom}(t_1) \cap \mathrm{dom}(t_2) = \emptyset$ and $\mathrm{dom}'(t_1) \cap \mathrm{dom}'(t_2) = \emptyset$. In this case, we write $t_1 \otimes t_2$ for

the partial bijection given by

$$t_1 \otimes t_2 = (\ \mathrm{dom}(t_1) \cup \mathrm{dom}(t_2),\ \mathrm{dom}'(t_1) \cup \mathrm{dom}'(t_2),\ t_1 \cup t_2)$$

Partial bijections are ordered as follows: $t' \geq t$ if and only if $t' = t \otimes t''$ for some (uniquely determined) t''.

5.2 Parameters

For aspect (b) of parameters we introduce a special symbol $\tau \notin \mathit{Regs}$ to represent the part of the store arising from regions "masked out" by the masking rule. Commands must not alter this portion of the store at all. We will thus sometimes refer to τ as the *silent* region. This intended meaning will become clear subsequently; for now, τ is just a symbol.

We are now ready to give a formal definition of parameters.

Definition 2 (parameter)**.** *A* parameter φ *is a pair* (Π, f) *where*

- Π *is a finite set of regions,*
- f *is a function assigning to each region* $\mathsf{r} \in \Pi \cup \{\tau\}$ *a partial bijection* $f(\mathsf{r})$ *such that distinct regions map to disjoint partial bijections.*
- $f(\tau) = (L, L', \emptyset)$ *for some* L, L'.

A parameter is meant to lay out regions in pairwise disjoint parts of the store in two related computations. The partial bijections relate corresponding locations and are used to interpret equality of locations. Note that $\varphi(\tau)$ merely contains two sets of locations and no "links" at all. The silent region represents a store portion that must not be modified at all.

If $\varphi = (\Pi, f)$ is a parameter, we refer to its components as follows:

$$\begin{array}{l} \mathrm{regs}(\varphi) = \Pi \\ \varphi(\mathsf{r}) = \begin{cases} f(\mathsf{r}) & \text{when defined} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathrm{dom}(\varphi) = \bigcup_{\mathsf{r} \in \Pi \cup \{\tau\}} \mathrm{dom}(\varphi(\mathsf{r})) \\ \mathrm{dom}'(\varphi) = \bigcup_{\mathsf{r} \in \Pi \cup \{\tau\}} \mathrm{dom}'(\varphi(\mathsf{r})) \end{array}$$

Note that $\mathrm{dom}(\varphi)$ and $\mathrm{dom}'(\varphi)$ are both finite. We define $\mathrm{Par}(\Pi) = \{\varphi \mid \mathrm{regs}(\varphi) \subseteq \Pi\}$. We also use the notation $\Pi \vdash \varphi\ ok$ to mean $\varphi \in \mathrm{Par}(\Pi)$. If φ and φ' are parameters such that

$$\mathrm{dom}(\varphi) \cap \mathrm{dom}(\varphi') = \mathrm{dom}'(\varphi) \cap \mathrm{dom}'(\varphi') = \emptyset$$

then φ, φ' are called *disjoint* and we write $\varphi \otimes \varphi'$ for the obvious juxtaposition of φ and φ'. Formally,

$$(\Pi, f) \otimes (\Pi', f') \ = \ (\Pi \cup \Pi', \lambda \mathsf{r}.(f\mathsf{r}) \otimes (f'\mathsf{r})) \ \in \mathrm{Par}(\Pi \cup \Pi')$$

Whenever we write $\varphi \otimes \psi$ then φ and ψ are presumed to be disjoint from each other so, a statement like $\exists \psi. \ldots \varphi \otimes \psi \ldots$ is understood as "there exists ψ disjoint from φ such that $\ldots \varphi \otimes \psi \ldots$".

Each set $\mathrm{Par}(\Pi)$ is partially ordered by $\varphi' \geq_\Pi \varphi \iff \varphi' = \varphi \otimes \psi$ for some ψ.

If t is a partial bijection then $[\mathsf{r} \mapsto t]$ is the parameter $\varphi \in \mathrm{Par}(\{\mathsf{r}\})$ such that $\varphi(\mathsf{r}) = t, \varphi(\mathsf{r}') = \emptyset$ when $\mathsf{r}' \neq \mathsf{r}$.

Thus, if $\ell \notin \mathrm{dom}(\psi)$ and $\ell' \notin \mathrm{dom}'(\psi)$ then we can form $\psi \otimes [\mathsf{r} \mapsto (\ell, \ell')]$ to add the link (ℓ, ℓ') to r in ψ. Similarly, if $\mathsf{r} \notin \Pi$, we can form $\varphi \otimes [\mathsf{r} \mapsto \emptyset]$ to initialise a new region r with $\emptyset$.

If $\Pi, \mathsf{r} \vdash \varphi\ ok$ then $\varphi - \mathsf{r}$ denotes the parameter such that $\Pi \vdash \varphi - \mathsf{r}\ ok$ and

$$\begin{array}{l} (\varphi - \mathsf{r})(\mathsf{r}') = \varphi(\mathsf{r}') \text{ when } \mathsf{r}' \neq \mathsf{r} \\ (\varphi - \mathsf{r})(\tau) = \varphi(\tau) \otimes (\mathrm{dom}(\varphi(\mathsf{r})), \mathrm{dom}'(\varphi(\mathsf{r})), \emptyset) \end{array}$$

5.3 Store relations

The main ingredient of our semantics for effects is the preservation of certain sets of relations on stores. We start by introducing the notion of relations on stores that depend only on particular subsets of locations in their domain and codomain.

Definition 3 (store relations)**.** *If* L, L' *are sets of locations, a* store relation *on* L, L' *is defined as a nonempty relation* $R \subseteq \mathbb{S} \times \mathbb{S}$ *such that whenever* $(s, s') \in R$ *and* $s \sim_L s_1$ *and* $s' \sim_{L'} s_1'$ *then* $(s_1, s_1') \in R$*, too. We write* $\mathrm{StRel}(L, L')$ *for the set of all store relations on* L, L'.

Given such a relation, we now formalize what it means to 'respect' an effect ε under some parameter φ.

Definition 4 (relations and effects)**.** *Suppose* $\Pi \vdash \varphi\ ok$ *and* $\Pi \vdash \varepsilon\ ok$*. Let* R *be a store relation on* $\mathrm{dom}(\varphi), \mathrm{dom}'(\varphi)$*. We say that* R respects ε at φ *if it is preserved by all commands that exhibit only* ε *on the store layout delineated by* φ*. Formally, we define:*

- R *respects* $\{rd_\mathsf{r}\}$ *at* φ *if* $(s, s') \in R$ *implies* $s.\ell = s'.\ell'$ *for all* $(\ell, \ell') \in \varphi(\mathsf{r})$*;*
- R *respects* $\{wr_\mathsf{r}\}$ *at* φ *if for all* $(s, s') \in R$ *and for all* $(\ell, \ell') \in \varphi(\mathsf{r})$ *and* $v \in \mathbb{Z}$*, we have* $(s[\ell \mapsto v], s'[\ell' \mapsto v]) \in R$*;*
- R *respects* $\{al_\mathsf{r}\}$ *always.*

We then define the set $\mathcal{R}_\varepsilon^\Pi(\varphi)$ *of all store relations that respect* ε *at* φ *as follows:*

$$\mathcal{R}_\varepsilon^\Pi(\varphi) = \{R \in \mathrm{StRel}(\mathrm{dom}(\varphi), \mathrm{dom}'(\varphi)) \mid \forall e \in \varepsilon, R \text{ resp. } e \text{ at } \varphi\}.$$

Unfortunately, we cannot track the allocation effect with relations; this will be done separately in the definition of the monad.

Finally, we introduce two additional bits of notation. If $s, s' \in \mathbb{S}$ we define

$$s, s' \models \varphi \iff \mathrm{dom}(s) = \mathrm{dom}(\varphi) \wedge \mathrm{dom}(s') = \mathrm{dom}'(\varphi)$$

We also define the following:

$$s \sim_\varphi s' \iff \forall \mathsf{r} \in \mathit{Regs}.\, \forall (\ell, \ell') \in \varphi(\mathsf{r}).\ s.\ell = s'.\ell'$$

Note that this notation does not constrain the values in the silent region.

6. Logical Relation

This section defines the relational semantics of refined types. It thus contains the main technical contribution of the paper.

Definition 5 (logical relation)**.** For $\Pi \vdash X\ ok$ and $\Pi \vdash \varphi\ ok$ we define a relation $[\![X]\!]_\varphi^\Pi \subseteq [\![U(X)]\!] \times [\![U(X)]\!]$ by

$$\begin{array}{rcl} [\![X]\!]_\varphi^\Pi & \equiv & \Delta_{[\![U(X)]\!]} \text{ when } X \in \{\mathtt{int}, \mathtt{bool}, \mathtt{unit}\} \\ [\![\mathtt{ref}_\mathsf{r}]\!]_\varphi^\Pi & \equiv & \varphi(\mathsf{r}) \\ [\![X \times Y]\!]_\varphi^\Pi & \equiv & [\![X]\!]_\varphi^\Pi \times [\![Y]\!]_\varphi^\Pi \\ [\![X \to T_\varepsilon Y]\!]_\varphi^\Pi & \equiv & \{(f, f') \mid \forall\, \varphi' \geq_\Pi \varphi. \forall (x, x') \in [\![X]\!]_{\varphi'}^\Pi. \\ & & \qquad (f(x), f'(x')) \in [\![T_\varepsilon Y]\!]_{\varphi'}^\Pi\} \\ [\![T_\varepsilon X]\!]_\varphi^\Pi & \equiv & \{(f, f') \mid s, s' \models \varphi \Rightarrow \\ & & \forall\, R \in \mathcal{R}_\varepsilon^\Pi(\varphi). s\ R\ s' \Rightarrow s_1\ R\ s_1' \ \wedge \\ & & \exists \psi \in \mathrm{Par}(\mathrm{als}(\varepsilon)).\ s_1, s_1' \models \varphi \otimes \psi \ \wedge \\ & & s_1 \sim_\psi s_1' \wedge (v, v') \in [\![X]\!]_{\varphi \otimes \psi}^\Pi \\ & & \text{where } (s_1, v) = f\ s \text{ and } (s_1', v') = f'\ s'\} \end{array}$$

We define $[\![\Theta]\!]_\varphi^\Pi$ by $[\![\Theta]\!]_\varphi^\Pi \equiv \{(\gamma, \gamma') \mid \forall\ i.\ (\gamma(x_i), \gamma'(x_i)) \in [\![X_i]\!]_\varphi^\Pi\}$ where $\Theta = x_1 : X_1, \ldots, x_n : X_n$.

The definition of the logical relation on computation types deserves some explanation. First, it says that the store behaviour of

two related computations must respect all relations that are compatible with the declared effect, cf. [8]. Since these relations are completely unconstrained on the silent region τ, this implies in particular that the silent region may neither be read nor modified. The existential quantifier asserts a (disjoint) extension ψ of the current parameter φ which is to hold all newly allocated references, hence $\psi \in \mathrm{Par}(\mathrm{als}(\varepsilon))$, cf. Def. 2. The result values (v, v') are then required to be related with respect to the extended parameter $\varphi \otimes \psi$. Note that if v and v' contain newly allocated references then $(v, v') \in [\![X]\!]^{\Pi}_{\varphi}$ will in general not hold.

The semantics of value types is monotonic with respect to the ordering on parameters.

Lemma 2 (Monotonicity)**.** *Suppose* $\Pi \vdash X$ *ok. If* $\varphi' \geq_\Pi \varphi$ *then* $[\![X]\!]^{\Pi}_{\varphi'} \supseteq [\![X]\!]^{\Pi}_{\varphi}$.

Proof. By induction on X. □

Lemma 3 (Masking)**.** *Suppose* $\Pi \vdash X$ *ok and* $\Pi, \mathsf{r} \vdash \varphi$ *ok. Then* $[\![X]\!]^{\Pi,\mathsf{r}}_{\varphi} = [\![X]\!]^{\Pi}_{\varphi-\mathsf{r}}$*, and likewise* $[\![T_\varepsilon\, X]\!]^{\Pi,\mathsf{r}}_{\varphi} = [\![T_\varepsilon\, X]\!]^{\Pi}_{\varphi-\mathsf{r}}$.

Proof. By induction on X. The cases where X is a basic type or a product type are trivial and hence omitted. Note, though, that if $X = \mathtt{ref}_{\mathsf{r}'}$ then $\mathsf{r}' \neq \mathsf{r}$.

Case $X = X_1 \to T_\varepsilon\, X_2$**.** Suppose that $(f, f') \in [\![X]\!]^{\Pi,\mathsf{r}}_{\varphi}$ and that $\varphi' \geq_\Pi \varphi - \mathsf{r}$ and that $(x, x') \in [\![X_1]\!]^{\Pi}_{\varphi'}$. Write $\varphi' = (\varphi - \mathsf{r}) \otimes \theta$ and put $\psi = \varphi \otimes \theta$. We have $\psi \geq_{\Pi,\mathsf{r}} \varphi$ and $(x, x') \in [\![X_1]\!]^{\Pi,\mathsf{r}}_{\psi}$ by the induction hypothesis since $\psi - \mathsf{r} = \varphi'$. Note that $\mathrm{als}(\varepsilon) \subseteq \Pi$ and so $\Pi \vdash \theta$ *ok*. The assumption then gives $(f\ v, f'\ v') \in [\![T_\varepsilon\, X_2]\!]^{\Pi,\mathsf{r}}_{\psi}$ and thence $(f\ v, f'\ v') \in [\![T_\varepsilon\, X_2]\!]^{\Pi}_{\varphi'}$ as required, again by the induction hypothesis and $\psi - \mathsf{r} = \varphi'$.

Conversely, assume that $(f, f') \in [\![X]\!]^{\Pi}_{\varphi-\mathsf{r}}$ and that $\varphi' \geq_{\Pi,\mathsf{r}} \varphi$ and $(x, x') \in [\![X_1]\!]^{\Pi,\mathsf{r}}_{\varphi'}$. We have $\varphi' - \mathsf{r} \geq_\Pi \varphi - \mathsf{r}$ and $(x, x') \in [\![X_1]\!]^{\Pi}_{\varphi'-\mathsf{r}}$ by the induction hypothesis. Therefore, the assumption gives us $(f\ x, f'\ x') \in [\![T_\varepsilon\, X_2]\!]^{\Pi}_{\varphi'-\mathsf{r}}$ whence $(f\ x, f'\ x') \in [\![T_\varepsilon\, X_2]\!]^{\Pi,\mathsf{r}}_{\varphi'}$ again by the induction hypothesis and we are done.

Case $T_\varepsilon(X)$**.** Suppose that $(f, f') \in [\![T_\varepsilon(X)]\!]^{\Pi,\mathsf{r}}_{\varphi}$ and $R \in \mathcal{R}^{\Pi}_{\varepsilon}(\varphi-\mathsf{r})$ (note that $\Pi \vdash \varepsilon$ *ok*) and $s, s' \models \varphi-\mathsf{r}$ and $(s, s') \in R$. Write $s_1, v = f\ s$ and $s_1', v' = f'\ s'$. We also have $s, s' \models \varphi$ by definition of $\varphi-\mathsf{r}$. Now, $R \in \mathcal{R}^{\Pi,\mathsf{r}}_{\varepsilon}(\varphi)$ since the masked out region r is not mentioned in ε; therefore the assumption provides us with $\psi \in \mathrm{Par}(\mathrm{als}(\varepsilon))$ disjoint from s, s' such that $s_1, s_1' \models \varphi'$ and $(s_1, s_1') \in R$ and $s_1 \sim_\psi s_1'$ and $(v, v') \in [\![X]\!]^{\Pi,\mathsf{r}}_{\varphi'}$ where $\varphi' = \varphi \otimes \psi$. Now put $\varphi'' = (\varphi-\mathsf{r}) \otimes \psi = \varphi'-\mathsf{r}$. We have $s_1, s_1' \models \varphi''$ and $(v, v') \in [\![X]\!]^{\Pi}_{\varphi''}$ by the induction hypothesis and we are done.

Conversely, suppose that $(f, f') \in [\![T_\varepsilon(X)]\!]^{\Pi}_{\varphi-\mathsf{r}}$ and $R \in \mathcal{R}^{\Pi,\mathsf{r}}_{\varepsilon}(\varphi)$ and $s, s' \models \varphi$ and $(s, s') \in R$. Again, write $s_1, v = f\ s$ and $s_1', v' = f'\ s'$. We also have $s, s' \models \varphi - \mathsf{r}$ by definition of $\varphi - \mathsf{r}$.

Now, as before, $R \in \mathcal{R}^{\Pi}_{\varepsilon}(\varphi - \mathsf{r})$ and so the assumption provides us with $\psi \in \mathrm{Par}(\mathrm{als}(\varepsilon))$ such that $s_1, s_1' \models \varphi'$ and $(s_1, s_1') \in R$ and $(v, v') \in [\![X]\!]^{\Pi}_{\varphi'}$ where $\varphi' = (\varphi-\mathsf{r}) \otimes \psi$.

Putting $\varphi'' = \varphi \otimes \psi$ it follows that $(v, v') \in [\![X]\!]^{\Pi,\mathsf{r}}_{\varphi''}$ from the induction hypothesis (note that $\varphi''-\mathsf{r} = \varphi'$). □

Lemma 4 (Extension)**.**

$$\mathcal{R}^{\Pi}_{\varepsilon-\mathsf{r}}(\varphi) = \mathcal{R}^{\Pi,\mathsf{r}}_{\varepsilon}(\varphi \otimes [\mathsf{r} \mapsto \emptyset])$$

Proof. The $\supseteq$ direction is obvious. Conversely, if R respects $\varepsilon - \mathsf{r}$ at φ then it also respects ε at $\varphi \otimes [\mathsf{r} \mapsto \emptyset]$ since the additional restrictions imposed by possible occurrences of r in ε are vacuously true at $\varphi \otimes [\mathsf{r} \mapsto \emptyset]$. □

The following establishes semantic soundness for our subtyping relation.

Lemma 5 (Soundness of subtyping)**.** *If* $\Pi \vdash X_i$ *ok and* $X_1 \leq X_2$ *then for all* $\varphi \in \mathrm{Par}(\Pi)$ *one has* $[\![X_1]\!]^{\Pi}_{\varphi} \subseteq [\![X_2]\!]^{\Pi}_{\varphi}$.

Proof. By induction on the subtyping derivation. □

We can now prove the following 'fundamental theorem' of logical relations, which states that terms are related to themselves.

Theorem 1 (Fundamental Theorem)**.**

1. *If* $\Pi \vdash \varphi$ *ok,* $\Pi; \Theta \vdash V : X$ *and* $(\gamma, \gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$*, then*
$$([\![U(\Theta) \vdash V : U(X)]\!]\gamma, [\![U(\Theta) \vdash V : U(X)]\!]\gamma') \in [\![X]\!]^{\Pi}_{\varphi}.$$
2. *If* $\Pi \vdash \varphi$ *ok,* $\Pi; \Theta \vdash M : T_\varepsilon(X)$ *and* $(\gamma, \gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$*, then*
$$([\![U(\Theta) \vdash M : T(U(X))]\!]\gamma, [\![U(\Theta) \vdash M : T(U(X))]\!]\gamma') \in [\![T_\varepsilon(X)]\!]^{\Pi}_{\varphi}.$$

Proof. By induction on typing derivations. We give a selection of representative cases.

Case `let`**.** From the derivation we have

$$\Pi; \Theta \vdash M_1 : T_{\varepsilon_1}(X_1) \tag{1}$$
$$\Pi; \Theta, x{:}X_1 \vdash M_2 : T_{\varepsilon_2}(X_2) \tag{2}$$

Assume $\Pi \vdash \varphi$ *ok* and $(\gamma, \gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$. Suppose $R \in \mathcal{R}^{\Pi}_{\varepsilon_1 \cup \varepsilon_2}(\varphi)$ and $s_0 R s_0'$. Let

$$f_1 := [\![U(\Theta) \vdash M_1 : TUX_1]\!],$$

and $(s_1, v_1) = f_1 \gamma s_0$ and $(s_1', v_1') = f_1 \gamma' s_0'$.

$$f_2 := [\![U(\Theta), x{:}U(X_1) \vdash M_2 : TUX_2]\!],$$

and $(s_2, v_2) = f_2\,(\gamma, v_1)\,s_1$ and $(s_2', v_2') = f_2\,(\gamma', v_1')s_1'$.
We then have $[\![U(\Theta) \vdash M : TUX_2]\!]\gamma s_0 = (s_2, v_2)$ and $[\![U(\Theta) \vdash M : TUX_2]\!]\gamma' s_0' = (s_2', v_2')$.
Now $R \in \mathcal{R}^{\Pi}_{\varepsilon_i}(\varphi)$ holds for $i = 1, 2$. The induction hypothesis for (1), applied to φ, (γ, γ') and (s_0, s_0') therefore yields $s_1 R s_1'$ and also furnishes $\psi_1 \in \mathrm{Par}(\mathrm{als}(\varepsilon_1))$ such that $s_1, s_1' \models \varphi \otimes \psi_1$ and $s_1 \sim_{\psi_1} s_1'$ and $(v_1, v_1') \in [\![X_1]\!]^{\Pi}_{\varphi'}$ where $\varphi' = \varphi \otimes \psi_1$.
We now define a relation R^* by

$$s R^* s' \iff s R s' \wedge s \sim_{\psi_1} s'$$

We then have $R^* \in \mathcal{R}^{\Pi}_{\varepsilon_2}(\varphi')$ and also $s_1 R^* s_1'$. In particular, R^* respects rd_{r} at ψ_1 in view of the clause $s \sim_{\psi_1} s'$ that has been added for that purpose.
The monotonicity of the interpretation of the value types (Lemma 2) yields $((\gamma, v_1), (\gamma', v_1')) \in [\![\Theta, x{:}X_1]\!]^{\Pi}_{\varphi'}$. We can therefore apply the induction hypothesis for (2), with φ', $((\gamma, v_1), (\gamma', v_1'))$ and (s_1, s_1'), and obtain the existence of some $\psi_2 \in \mathrm{Par}(\mathrm{als}(\varepsilon_2))$ such that $s_2, s_2' \models \varphi' \otimes \psi_2$ and $s_2\, R^*\, s_2'$ and $s_2 \sim_{\psi_2} s_2'$ and $(v_2, v_2') \in [\![X_2]\!]^{\Pi}_{\varphi \otimes \psi_1 \otimes \psi_2}$. From the definition of R^* we then get $s_2 R s_2'$ and $s_2 \sim_{\psi_1 \otimes \psi_2} s_2'$.

Case `ref`**.** Suppose $\Pi \vdash \varphi$ *ok*, $(\gamma, \gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$, $R \in \mathcal{R}^{\Pi}_{al_{\mathsf{r}}}(\varphi)$, i.e., arbitrary and $s_0 R s_0'$. Writing

$$f := [\![U\Theta \vdash \mathtt{ref}(V) : T\,\mathtt{ref}]\!]$$

we have

$$f\gamma s = new(s, \gamma(x)) = (s_1, \ell) \quad \text{and}$$
$$f\gamma' s' = new(s', \gamma'(x)) = (s_1', \ell').$$

Now, since R is a store relation on $\mathrm{dom}(\varphi), \mathrm{dom}'(\varphi)$ and $s_0, s_0' \models \varphi$ we find $s_1 R s_1'$. We let $\psi = [\mathsf{r}\mapsto(\ell,\ell')]$ and have $s_1, s_1' \models \varphi \otimes \psi$. We have $(\ell,\ell') \in [\![\mathtt{ref}_{\mathsf{r}}]\!]^{\Pi}_{\varphi\otimes\psi}$ and since $[\![U\Theta \vdash V : \mathtt{int}]\!]\gamma = [\![U\Theta \vdash V : \mathtt{int}]\!]\gamma'$ from $(\gamma,\gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$, we also have $s_1 \sim_{\psi} s_1'$ and we are done.

Masking Rule Suppose $\Pi \vdash \Theta\ ok$, $\Pi \vdash X\ ok$, $\Pi, \mathsf{r}; \Theta \vdash M : T_\varepsilon X$, $\Pi \vdash \varphi\ ok$, and $(\gamma,\gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$ and $\Pi \vdash \varphi\ ok$ and $R \in \mathcal{R}^{\Pi}_{\varepsilon-\mathsf{r}}(\varphi)$ and $s_0 R s_0'$ and $s_0, s_0' \models \varphi$.

Write $g := [\![U\Theta \vdash M : TUX]\!]$ and $(s_1, v_1) = g\gamma s_0$ and $(s_1', v_1') = g\gamma s_0'$.

Let us write $\psi = \varphi \otimes [\mathsf{r}\mapsto\emptyset]$. The masking lemma 3 shows $(\gamma,\gamma') \in [\![\Theta]\!]^{\Pi,\mathsf{r}}_{\psi}$ and from the extension lemma 4 we have $R \in \mathcal{R}^{\Pi,\mathsf{r}}_{\varepsilon}(\psi)$. So the induction hypothesis furnishes $\theta \in \mathrm{Par}(\mathrm{als}(\varepsilon))$ and disjoint from s_0, s_0' such that $s_1, s_1' \models \psi'$ and $s_1 \sim_\theta s_1'$ and $s_1 R s_1'$ and $(v_1, v_1') \in [\![X]\!]^{\Pi,\mathsf{r}}_{\psi'}$ where $\psi' = \psi \otimes \theta$. Now, $\psi' - \mathsf{r} = \varphi \otimes (\theta - \mathsf{r})$ and so $(v_1, v_1') \in [\![X]\!]^{\Pi}_{\varphi\otimes(\theta-\mathsf{r})}$ again by the masking lemma. The rest is clear from the definitions.

□

Definition 6 (contextual equivalence)**.**

1. *Suppose that* $\Pi; \emptyset \vdash V_i : X$ *for* $i = 1, 2$ *are two closed values. They are* contextually equivalent*, written* $\Pi; \emptyset \vdash V_1 \equiv V_2 : X$*, if for all values* $\Pi; \emptyset \vdash V : X \to T_\varepsilon\, \mathtt{bool}$ *("contexts") with* ε *arbitrary it holds that when* $(s_1, v_1) = [\![V\ V_1]\!]\ ()\ \emptyset$ *and* $(s_2, v_2) = [\![V\ V_2]\!]\ ()\ \emptyset$ *then* $v_1 = v_2$*. Recall that* $\emptyset$ *is the initial, empty state.*
2. *Contextual equivalence is extended to open values and computations by abstracting all free variables using a dummy abstraction of type* `unit` *in the case of a closed computation. We write* $\Pi; \Theta \vdash M_1 \equiv M_2 : A$ *to mean that computations* M_1 *and* M_2 *are contextually equivalent.*

Notice that the examining context V must itself be typable in our effect system. This is important when, e.g., $X = (\mathtt{int} \to T_\emptyset\, \mathtt{unit}) \to T_\varepsilon\, \mathtt{unit}$ and the equivalence of V_1 and V_2 relies on their being fed a pure function as input. E.g., in this situation we would want to consider $V_1 = \lambda f.f(0)$ and $V_2 = \lambda f.f(0); f(0)$ as equivalent.

This definition coincides with the more standard yet more complex one involving terms with holes as contexts, cf. [8]. This is due to the fact that we use a monadic metalanguage rather than a language with built-in side effects like ML.

We remark that the restriction to boolean observations is not a severe one. Two contextually equivalent values also agree on all observations of type `int` as is easily seen by wrapping the observation into an appropriate equality test.

Proposition 1. *Contextual equivalence is a congruence validating the equational theory of the monadic metalanguage (in particular beta reduction).*

The following corollary to Theorem 1 now provides a powerful method for establishing stronger contextual equivalences.

Corollary 1. *Suppose that* $\Pi; \Theta \vdash M_i : T_\varepsilon\, X$ *for* $i = 1, 2$ *and whenever* $\Pi \vdash \varphi\ ok$ *and* $(\gamma,\gamma') \in [\![\Theta]\!]^{\Pi}_{\varphi}$ *then*

$$([\![U(\Theta) \vdash M_1 : T(U(X))]\!]\gamma, [\![U(\Theta) \vdash M_2 : T(U(X))]\!]\gamma') \in [\![T_\varepsilon\, X]\!]^{\Pi}_{\varphi}.$$

Then M_1 *and* M_2 *are contextually equivalent.*

Proof. Let $\lambda(M_1)$ and $\lambda(M_2)$ be the values obtained by lambda abstracting all variables in M_1, M_2. Let us write $\Theta \to T_\varepsilon\, A$ for the common type of these values. It is easy to see from the definition of the logical relation that

$$([\![\lambda(M_1)]\!], [\![\lambda(M_2)]\!]) \in [\![\Theta \to T_\varepsilon\, X]\!]^{\Pi}_{\varphi}$$

for all φ (we have omitted types and contexts here). Now let $V : (\Theta \to T_\varepsilon\, X) \to T_{\varepsilon'}\, \mathtt{bool}$ be a context. By applying the fundamental lemma to V we conclude

$$([\![V\ \lambda(M_1)]\!], [\![V\ \lambda(M_2)]\!]) \in [\![T_{\varepsilon'}\, \mathtt{bool}]\!]^{\Pi}_{\varphi}$$

Now, let φ be the parameter that assigns to each region the empty partial bijection. We have $\emptyset, \emptyset \models \varphi$ where $\emptyset$ is the empty state. Let $R \in \mathcal{R}^{\Pi}_{\varepsilon'}(\varphi)$ be arbitrary. Clearly, $\emptyset R \emptyset$ no matter what ε' is. Put $(s_i, v_i) = [\![V(\lambda(M_i))]\!]$. By the definition of the logical relation we then have $v_1 = v_2$ as required. □

7. Applications

For $\Pi \vdash \varphi\ ok$ and $\Pi \vdash \varepsilon\ ok$ and $s, s' \models \varphi$ we introduce the notation

$$s \sim_{\mathrm{rds}_\varphi(\varepsilon)} s' \iff \forall \mathsf{r} \in \mathrm{rds}(\varepsilon).\forall(\ell,\ell') \in \varphi(\mathsf{r}).s.\ell = s'.\ell'$$

It expresses that s and s' agree on those locations that are read given effect ε.

We also define

$$nwrs_\varphi(\varepsilon) = \mathrm{dom}(\varphi) \setminus \textstyle\bigcup_{\mathsf{r}\in\mathrm{wrs}(\varepsilon)} \mathrm{dom}(\varphi(\mathsf{r}))$$
$$nwrs'_\varphi(\varepsilon) = \mathrm{dom}'(\varphi) \setminus \textstyle\bigcup_{\mathsf{r}\in\mathrm{wrs}(\varepsilon)} \mathrm{dom}'(\varphi(\mathsf{r}))$$

Thus $nwrs_\varphi(\varepsilon)$ and $nwrs'_\varphi(\varepsilon)$ comprise the locations on the left (resp. right) side that are not written to given effect ε. This includes the locations in the silent region.

Lemma 6. *Suppose* $\Pi; \Theta \vdash M : T_\varepsilon X$ *and* $\Pi \vdash \varphi\ ok$ *and* $(\gamma,\gamma') \in [\![\Theta]\!]_\varphi$ *and* $s_0, s_0' \models \varphi$ *and* $[\![M]\!]\gamma s_0 = (s_1, x)$ *and* $[\![M]\!]\gamma' s_0' = (s_1', x')$.

If $s_0 \sim_{\mathrm{rds}_\varphi(\varepsilon)} s_0'$ *then there exists* $\psi \in \mathrm{Par}(\mathrm{als}(\varepsilon))$ *disjoint from* s_0, s_0' *such that*

1. $s_1, s_1' \models \varphi \otimes \psi$ *and* $(x, x') \in [\![X]\!]_{\varphi\otimes\psi}$ *and* $s_1 \sim_\psi s_1'$.
2. $s_0 \sim_{nwrs_\varphi(\varepsilon)} s_1$ *and* $s_0' \sim_{nwrs'_\varphi(\varepsilon)} s_1'$.
3. *For each* $(\ell,\ell') \in \varphi(\mathsf{r})$ *where* $\mathsf{r} \in \Pi$ *we have either*
   - $s_0.\ell = s_1.\ell$ *and* $s_0'.\ell' = s_1'.\ell'$ *(unchanged) or*
   - $s_1.\ell = s_1'.\ell'$ *(identically written).*
4. *Suppose that* $\ell \in \mathrm{dom}(\varphi)$ *but there is no* ℓ', r *such that* $(\ell,\ell') \in \varphi(\mathsf{r})$*. Then* $s_0.\ell = s_1.\ell$*. A symmetric statement holds for* s_0', s_1'.

Notice that Part 2 asserts in particular that the contents of the silent region do not change from s_0 to s_1.

Proof. Part 1 is direct from the definition.

For part 2 we define

$$(s, s') \in R \iff s \sim_{\mathrm{rds}_\varphi(\varepsilon)} s' \wedge s \sim_{nwrs_\varphi(\varepsilon)} s_0$$

We now have $s_0 R s_0'$ (not necessarily $s_0 R s_0'$) and so $s_1 R s_1'$. The claim about s_1 follows. The proof for $nwrs'_\varphi(\varepsilon)$ and s_1' is analogous.

For part 3 we define

$$(s, s') \in R \iff s \sim_{\mathrm{rds}_\varphi(\varepsilon)} s' \wedge \forall \mathsf{r}\in\Pi.\forall(\ell,\ell') \in \varphi(\mathsf{r}).\ s.\ell = s_0.\ell \wedge s'.\ell' = s_0'.\ell' \vee s.\ell = s'.\ell'$$

and note that $R \in \mathcal{R}^{\Pi}_{\varepsilon}(\varphi)$. It follows $s_1, s_1' \in R$ and the claim follows.

Part 4, finally, follows using the relation

$$s R s' \iff s \sim_{\mathrm{rds}_{\varphi\varepsilon}} s' \wedge s.\ell = s_0.\ell$$

□

Proposition 2 (duplicated computation)**.** *Suppose that* $\Pi; \Theta \vdash M : T_\varepsilon(X)$ *and suppose that* $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \mathrm{als}(\varepsilon) = \emptyset$*. Thus,*

M reads and writes on disjoint portions of the store and makes no allocations except possibly in the silent region. Then the following two terms are contextually equivalent:

$$M_1 := \texttt{let}\ x{\Leftarrow}M\ \texttt{in}\ \texttt{val}\ (x,x)$$
$$M_2 := \texttt{let}\ x{\Leftarrow}M\ \texttt{in}\ \texttt{let}\ y{\Leftarrow}M\ \texttt{in}\ \texttt{val}\ (x,y)$$

Formally, $\Pi;\Theta \vdash M_1 \equiv M_2 : T_\varepsilon\,(X \times X)$.

Proof. We will use Cor. 1. Let $f(\gamma) = [\![U\Theta \vdash M : TUX]\!]\gamma$. Suppose that $\Pi \vdash \varphi\ ok$ and $(\gamma,\gamma') \in [\![\Theta]\!]^\Pi_\varphi$. Suppose that $s_0, s_0' \models \varphi$ and define

$$s_1, v_1 = f(\gamma)(s_0)$$
$$s_2, v_2 = f(\gamma)(s_1)$$
$$s_1', v_1' = f(\gamma')(s_0')$$

Suppose that $s_0 R s_0'$ for some $R \in \mathcal{R}^\Pi_\varepsilon(\varphi)$. We must exhibit some $\psi \in \mathrm{Par}(\emptyset)$ disjoint from s, s' such that $s_2, s_1' \models \varphi \otimes \psi$ and $(v_2, v_1') \in [\![X]\!]^\Pi_{\varphi\otimes\psi}$ and $(v_1, v_1') \in [\![X]\!]^\Pi_{\varphi\otimes\psi}$ and $s_2 \sim_\psi s_1'$ and $s_2 R s_1'$. Notice that ψ contains allocations in the silent region only.

From $s_0 R s_0'$ it follows that $s_0 \sim_{\mathrm{rds}_\varphi(\varepsilon)} s_0'$. Thus, Lemma 6 furnishes $\psi_1 \in \mathrm{Par}(\emptyset)$ disjoint from s_0, s_0' such that $s_1, s_1' \models \varphi \otimes \psi_1$ and $(v_1, v_1') \in [\![X]\!]^\Pi_{\varphi\otimes\psi_1}$. Furthermore, since $\mathrm{rds}(\varphi) \cap \mathrm{wrs}(\varphi) = \emptyset$ we have $s_1 \sim_{\mathrm{rds}_\varphi(\varepsilon)} s_0'$. Decompose ψ_1 as $\psi_1^L \otimes \psi_1^R$ where $\mathrm{dom}'(\psi_1^L) = \emptyset = \mathrm{dom}(\psi_1^R)$.

Now we have $s_1, s_0' \models \varphi \otimes \psi_1^L$ and thus another application of Lemma 6 furnishes $\psi_2 \in \mathrm{Par}(\emptyset)$ disjoint from s_1, s_0' such that $s_2, s_1' \models \varphi \otimes \psi_1^L \otimes \psi_2$ and $(v_2, v_1') \in [\![X]\!]^\Pi_{\varphi\otimes\psi_1^L\otimes\psi_2}$.

Again decompose ψ_2 as $\psi_2 = \psi_2^L \otimes \psi_2^R$ where $\mathrm{dom}'(\psi_2^L) = \emptyset = \mathrm{dom}(\psi_2^R)$.

We already know from Part 2 of Lemma 6 that if location ℓ is not written (that includes locations in the silent region) then $s_1.\ell = s_2.\ell$. We claim that s_1 agrees with s_2 on the other, written to, locations as well. To see this, we define s_0^* by updating s_0 in such a way that for all $\mathsf{r} \in \Pi$ and $(\ell,\ell') \in \varphi(\mathsf{r})$ one has $s_0^*.\ell' = s_0.\ell$ and s_0^* agrees with s_0' elsewhere. Then $s_1, s_0^* \models \varphi \otimes \psi_1^L$ and $s_1 \sim_{\mathrm{rds}_\varphi(\varepsilon)} s_0^*$. Let s_1^* denote the successor state of s_0^*. Lemma 6,3 yields that for each $(\ell,\ell') \in \varphi(\mathsf{r})$ that either $s_2.\ell = s_1.\ell$ or $s_2.\ell = s_1^*.\ell'$. On the other hand, $s_1^*.\ell' = s_1.\ell$ or $s_1.\ell = s_0.\ell$ and $s_1^*.\ell' = s_0^*.\ell'$. It follows $s_2.\ell = s_1.\ell$. We have thus shown $s_1 \sim_{\mathrm{dom}(\varphi)} s_2$. Since $s_1 R s_1'$ and R is a store relation on $\mathrm{dom}(\varphi), \mathrm{dom}'(\varphi)$ we conclude $s_2 R s_1'$ and we are done. □

Proposition 3 (commuting computations)**.** *Suppose that*

$$\Pi;\Theta \vdash M_1 : T_{\varepsilon_1}(X)$$
$$\Pi;\Theta \vdash M_2 : T_{\varepsilon_2}(X)$$

and suppose that $\mathrm{rds}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \mathrm{rds}(\varepsilon_2) \cap \mathrm{wrs}(\varepsilon_1) = \mathrm{wrs}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset$.

Put

$$N_1 := \texttt{let}\ y{\Leftarrow}M_1\ \texttt{in}\ \texttt{let}\ x{\Leftarrow}M_2\ \texttt{in}\ \texttt{val}\ (x,y)$$
$$N_2 := \texttt{let}\ x{\Leftarrow}M_2\ \texttt{in}\ \texttt{let}\ y{\Leftarrow}M_1\ \texttt{in}\ \texttt{val}\ (x,y)$$

We have

$$\Pi;\Theta \vdash N_1 \equiv N_2 : T_\varepsilon\,(X \times X)$$

where $\varepsilon = \varepsilon_1 \cup \varepsilon_2$.

Proof. We will use Cor. 1. Let $f_i(\gamma) = [\![U\Theta \vdash M_i : TUX]\!]\gamma$ for $i = 1, 2$.

Suppose that $\Pi \vdash \varphi\ ok$ and $(\gamma,\gamma') \in [\![\Theta]\!]^\Pi_\varphi$. Suppose that $s_0, s_0' \models \varphi$ and define

$$s_1, v_1 = f_1(\gamma)(s_0)$$
$$s_2, v_2 = f_2(\gamma)(s_1)$$
$$s_1', v_1' = f_2(\gamma')(s_0')$$
$$s_2', v_2' = f_1(\gamma')(s_1')$$

Suppose that $s_0 R s_0'$ for some $R \in \mathcal{R}^\Pi_\varepsilon(\varphi)$.

We must exhibit $\psi \in \mathrm{Par}(\mathrm{als}(\varepsilon))$ disjoint from s_0, s_0' such that $s_2, s_2' \models \varphi'$ and $(v_1, v_2') \in [\![X]\!]_{\varphi'}$ and $(v_2, v_1') \in [\![X]\!]_{\varphi'}$ where $\varphi' = \varphi \otimes \psi$ and $s_2 R s_2'$.

Lemma 6 applied to M_2 together with the fact that $\mathrm{rds}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset$ shows that $s_0 \sim_{\mathrm{rds}_\varphi(\varepsilon_1)} s_1'$. We also have $s_0, s_1' \models \varphi$ so Lemma 6 applied to M_1 furnishes ψ_1 disjoint from s_0, s_1' such that $(v_1, v_2') \in [\![X]\!]^\Pi_{\varphi\otimes\psi_1}$ and also $s_1, s_2' \models \varphi \otimes \psi_1$. Arguing symmetrically, we obtain ψ_2 disjoint from s_1, s_0' such that $(v_2, v_1') \in [\![X]\!]^\Pi_{\varphi\otimes\psi_2}$ and also $s_2, s_1' \models \varphi \otimes \psi_2$. Now, since $\mathrm{dom}(\psi_2)$ is disjoint from $\mathrm{dom}(s_1) \supseteq \mathrm{dom}(\psi_1)$ and analogously on the right-hand side, we find that ψ_1 and ψ_2 are disjoint allowing us to put $\psi := \psi_1 \otimes \psi_2$ and $(v_2, v_1'), (v_1, v_2') \in [\![X]\!]^\Pi_{\varphi\otimes\psi}$ by monotonicity. We also have $s_2, s_2' \models \varphi\otimes$.

Since $s_0 R s_0'$ we can show $s_2 R s_2'$ by induction on the size of $\mathrm{dom}_{le/ri}(\varphi)$ using the definition of $\mathcal{R}^\Pi_\varepsilon(\varphi)$. □

Proposition 4 (dead computation)**.** *Suppose that*

$$\Pi;\Theta \vdash M : T_\varepsilon\,(\texttt{unit})$$

and that $\mathrm{wrs}(\varepsilon) = \emptyset$. *Then M is contextually equivalent to* $\texttt{val}\ ()$.

Proof. Let $\varphi \in \mathrm{Par}(\Pi)$ and $(\gamma,\gamma') \in [\![\Theta]\!]_\varphi$. Let $f = [\![M]\!]\gamma$ and suppose $s_0, s_0' \models \varphi$. Let $v, s_1 = f s_0$. Of course, $v = \star$. The fundamental lemma applied to M, φ, s_0, s_0' furnishes $\psi \in \mathrm{Par}(\mathrm{als}(\varepsilon))$ disjoint from φ. Construct ψ_1 by taking only the left part of ψ; formally: $\psi_1(\mathsf{r}) = (\mathrm{dom}(\psi(\mathsf{r})), \emptyset, \emptyset)$. Now, $s_1, s_0' \models \varphi \otimes \psi_1$ and, trivially, $(v,\star) \in [\![\texttt{unit}]\!]_{\varphi\otimes\psi_1}$.

If $R \in \mathcal{R}^\Pi_\varepsilon(\varphi)$ and $s_0 R s_0'$ then s_1 agrees with s_0 on all locations that R depends upon, so $s_1 R s_0'$. □

Proposition 5 (pure lambda hoist)**.** *Suppose that*

$$\Pi;\Theta \vdash M : T_\emptyset\ Z$$
$$\Pi;\Theta, x{:}X, y{:}Z \vdash N : T_\varepsilon\ Y$$

Put

$$M_1 := \texttt{val}(\lambda x{:}UX.\texttt{let}\ y{\Leftarrow}M\ \texttt{in}\ N)$$
$$M_2 := \texttt{let}\ y{\Leftarrow}M\ \texttt{in}\ \texttt{val}(\lambda x{:}UX.N)$$

Then $\Pi;\Theta \vdash M_1 \equiv M_2 : T_\emptyset\,(X \to T_\varepsilon\,)$

Proof. Let $\varphi \in \mathrm{Par}(\Pi)$ and $(\gamma,\gamma') \in [\![\Theta]\!]_\varphi$ and define

$$f = [\![M]\!]\gamma$$
$$f' = [\![M]\!]\gamma'$$
$$g(v_x, v_y) = [\![N]\!]\gamma[x{\mapsto}v_x, y{\mapsto}v_y]$$
$$g'(v_x, v_y) = [\![N]\!]\gamma'[x{\mapsto}v_x, y{\mapsto}v_y]$$

Suppose that $s_0, s_0' \models \varphi$. The fundamental theorem applied to M yields $\psi_1 \in \mathrm{Par}(\emptyset)$ such that $s_1, s_1' \models \varphi \otimes \psi$ where $s_1, v = f s_0$ and $s_1', v' = s_0'$.

Factor ψ_1 as $\psi_1^L \otimes \psi_1^R$ such that $s_1, s_0' \models \varphi \otimes \psi_1^L$. We claim that

$$(\lambda v_x.g(v_x, v)\ , \lambda v_x.\lambda s'.\mathit{let}\ s_2', v_y{=}f's'\ \mathit{in}\ g'(v_x, v_y)s_2') \in [\![X \to T_\varepsilon]\!]_{\varphi\otimes\psi_1^L}$$

Assume thus $\varphi' \geq_\Pi \varphi \otimes \psi_1^L$ and $(v_x, v_x') \in [\![X]\!]_{\varphi'}$. Also assume $s, s' \models \varphi'$ and let $s_2', v_y = f's'$. We would like to argue that $(v, v_y) \in [\![Z]\!]_{\varphi'\otimes\psi_2}$ for some ψ_2.

Now $s_0, s' \models \varphi$ whence $(v, v_y) \in [\![Z]\!]_{\varphi\otimes\psi_1^L\otimes\theta}$ where θ is on the right hand side only. We also know from $s, s' \models \varphi'$ that θ is disjoint from φ'. Thus, we have $(v, v_x) \in [\![Z]\!]_{\varphi'\otimes\theta}$ by monotonicity.

The rest is similar to earlier calculations. □

Examples Suppose that the let bindings $f_1 \Leftarrow M_{\mathrm{buf}}$ and $f_2 \Leftarrow M_{\mathrm{buf}}$ are in force. If we type them using two different region identifiers Prop. 3 and Prop. 4 and Prop. 1 allow us to conclude that

$$\texttt{let}\ x \Leftarrow f_1(5)\ \texttt{in let}\ y \Leftarrow f_1(4)\ \texttt{in let}\ z \Leftarrow f_2(6)\ \texttt{in}\ (x, z) \equiv \texttt{let}\ z \Leftarrow f_2(6)\ \texttt{in let}\ x \Leftarrow f_1(5)\ \texttt{in}\ (x, z)$$

Lambda hoist (Prop. 5) applies to the following code after using the masking rule to give a pure typing to V_{sum}.

$$\lambda x.\texttt{let}\ s \Leftarrow V_{\mathrm{sum}}(a, b, c)\ \texttt{in val}\ (x + s)$$

We remark that the program equivalences we get for pure computations are complete in the following sense:

Proposition 6. *Let C be a cartesian closed category and T be a strong monad on C. Suppose that in the Kleisli category C_T the laws of dead computation, commuting computations, duplicated computations, lambda hoist are valid. Then C_T is cartesian closed.*

In particular, the Kleisli category consisting of computations of type $T_\emptyset(A)$ modulo contextual equivalence is cartesian closed.

8. Conclusion and further work

We have given a relational semantics to a region-based effect type system for a higher-order language with dynamically allocated store. The relational semantics is shown sound for contextual equivalence and thus provides a powerful proof principle for the latter. We have used the semantics to establish the soundness of a collection of useful effect-based program transformations. It would probably be very hard to establish these directly from the definition of contextual equivalence and no such proof appears to exist in the literature.

There has been a great deal of previous work on the soundness of region-based memory management and of its close cousin, encapsulated monadic state, as provided by `runST` in Haskell [15]. We mention some particularly relevant references. Banerjee et al. [3] translate the region calculus into a variant of System F and give a denotational model showing that references in masked regions do not affect the rest of the computation. Moggi and Sabry [20] prove syntactic type soundness for encapsulated lazy state. Fluet and Morrisett [12] bring the two lines of work together by giving a type- and behaviour-preserving translation from a variant of the region calculus into an extension of System F with a region-indexed family of monads. Naumann [21] uses simulation relations to capture a notion of observational purity for boolean-valued specifications that allows mutation of encapsulated state.

The general problem of modelling and proving equivalences in languages with dynamically allocated store and higher order features is a difficult one, with a very long history [27]. The basic techniques we use here, such as partial bijections and parametric logical relations, have been developed and refined over the last 25 years or so [14, 18, 22, 23, 24, 9]. The focus of much of this previous work has been on showing tricky equivalences between particular pairs of terms, such as the well-known Meyer-Sieber examples [18]. One might expect that equivalences justified by simple program analyses, such as those considered here, would generally be much easier to establish than some of the more contorted examples from the literature. Whilst this is broadly true – our relational reasoning technique is far from complete, yet suffices for establishing the interesting equational consequences of the effect system – completely generic reasoning is surprisingly difficult. When proving concrete equivalences one treats the context generically, but has two particular, literal terms in one's hand, whose denotations one can work with explicitly. In the case of purely type-based equivalences, on the other hand, both the context and the terms are abstract; all one knows are the types, and the semantics of those types has to capture enough information to justify all instances of the transformation.

An alternative approach to proving 'difficult' contextual equivalences is to use techniques based on bisimulation. Originally developed for process calculi by Park and Milner [19], bisimulation was adapted for the untyped lambda calculus by Abramsky [2]. Other researchers, particularly Sumii and Pierce, subsequently developed notions of bisimulation for typed lambda calculi that could deal with the kind of encapsulation (data abstraction) given by existential types [26]. These methods have recently been refined by Koutavas and Wand, and applied to an untyped higher-order language with storage [16] and to object-based calculi. It would be extremely worthwhile to investigate whether bisimulation methods can be applied to the typed (and, as discussed above, type-directed) impure equivalences studied here.[2]

It has been suggested to us that our results might formally be subsumed by our earlier work [8] via a translation that essentially treats a region as a global variable. Reading / writing within a region is then tracked as a read write from/to the corresponding global variable. First, this handles neither masking nor the allocation effect. More importantly such a translation would at best allow one to transfer a type *inference* for global variables to one for the region calculus but it does not seem to help in any way with the soundness proof.

Our base language is deliberately simple so as to allow us to focus on the salient aspects of the semantics. Nevertheless, it would be useful to extend it in various ways. We sketch how this can be done and what difficulties might be faced.

Recursion To accommodate recursive functions, one needs to phrase the denotational semantics in terms of cpos and partial functions. A new 'possible non-termination' effect must also be introduced (and attached to potentially-diverging recursive definitions), which prevents "dead computation" from applying, see [5]. The relations interpreting refined types must then be admissible in the sense that suprema of componentwise related ascending chains are related.

To enforce this condition, we must replace the current definition of $[\![T_\varepsilon\ X]\!]^{\Pi}_{\varphi}$ with the least admissible relation comprising it. This is because the existential quantification in the definition of this relation thwarts an attempt to show admissibility by induction on types. The least admissible relation $R^\dagger$ comprising a given relation R on a cpo D is explicitly given by the set of all (d, d') for which there exist chains $d = \sup_i d_i, d' = \sup_i d'_i$ where $(d_i, d'_i) \in R$ for all i. With this amended definition the cases go through at the expense of a slightly more cluttered notation.

The notational messiness is considerably palliated by the fact that when faced with a situation where one has an assumption of the form $dR^\dagger d'$ and one wants to show some property $\varphi(d, d')$ where φ is itself admissible then one can w.l.o.g. assume that one has dRd'.

An alternative route to admissibility consists of using a semantics phrased in terms of a continuation-based termination judgement [23, 9]. The existential quantifier in the monad then becomes a universal quantifier and admissibility holds from the start.[3] This approach, however, yields an apparently coarser relation which breaks our current proof of Prop 2. Writing $M_1 \simeq M_2$ to mean that the denotations of M_1 and M_2 are in the continuation-based logical relation, we are faced with the apparently simpler goal $(\texttt{x=}M\texttt{;y=}M\texttt{;(x,y)}) \simeq (\texttt{x=}M\texttt{;(x,x)})$, however, we have to prove it under the *weaker* assumption $M \simeq M$. This is an example of the difference between proving particular and generic equiv-

[2] The more general relationship between logical relations and bisimulation still seems slightly murky (at least to us) and clearly demands further study as well.

[3] In fact, rather more holds: the continuation-based formulation uses a '$\top\top$-closure' operation that 'extensionalizes' the relations [1].

alences discussed above: the problem does not arise in concrete cases in which we know what M actually is.

Purity Certain valid program equivalences are not provable with our proof method. For example, we would like the semantics to justify the pure typing

$$M_{\mathrm{mem}} : T_{\emptyset}\,(\mathtt{int} \to T_{\emptyset}\,\mathtt{int})$$

for M_{mem} and similar memoised functions on the grounds that they behave just as a pure function. With such a typing, an instance of M_{mem} could be hoisted out of an abstraction which it presently cannot. One way to make our semantics believe that M_{mem} is pure consists of including invariants on private portions of the store again as in [23, 9]. This would allow one to justify a strengthened masking rule with a semantically formulated side condition which requires one to "manually" provide an invariant on privately allocated store. In the case of the memo function M_{mem} one could use $\mathtt{read}(y) = \mathtt{read}(x) + 1$ as invariant. One has to prove that this invariant is maintained and moreover that the result value is independent of the store so long as the invariant is satisfied. At present, we cannot see a syntactic approximation to such a semantic masking rule, i.e., a non-contrived typing rule which would recognize M_{mem} as pure yet of course reject M_{buf} as impure.

General references Of course it would be interesting to include references to types other than integers. References to product and reference types which correspond to the usual Java/C heap model should be relatively straightforward and quite useful. Treating references to functions and values of general mixed-variance recursive types ought to be possible using a simultaneous recursive definition of the logical relation and the sets $\mathcal{R}^{\Pi}_{\varepsilon}(\varphi)$. While doing this is technically challenging, the principal feasibility of this approach has been demonstrated in [25, 11].

References

[1] M. Abadi. $\top\top$-closed relations and admissibility. *Mathematical Structures in Computer Science*, 10(3), 2000.

[2] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Directions in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, 1988.

[3] A. Banerjee, N. Heintze, and J. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS)*, 1999.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004. Revised version available from `http://research.microsoft.com/~nick/publications.htm`.

[5] N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *3rd ACM-SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2007.

[6] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS*. Springer-Verlag, 2002.

[7] N. Benton and A. Kennedy. Monads, effects and transformations. In *3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 26 of *ENTCS*. Elsevier, 1999.

[8] 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.

[9] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *LNCS*, 2005.

[10] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics, International Summer School, Springer LNCS 2395*, pages 42–122, 2000.

[11] N. Bohr and L. Birkedal. Relational reasoning for recursive types and references. In *APLAS*, 2006.

[12] M. Fluet and G. Morrisett. Monadic regions. *Journal of Functional Programming*, 2006. to appear.

[13] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, Massachusetts, August 1986.

[14] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, 1984.

[15] S. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4), 1995.

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

[17] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM Symposium on Principles of Programming Languages (POPL)*, 1988.

[18] A. R. Meyer and K. Sieber. Towards a fully abstract semantics for local variables: Preliminary report. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1988.

[19] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[20] E. Moggi and A. Sabry. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming*, 11(6), 2001.

[21] D. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, To appear.

[22] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[23] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.

[24] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.

[25] B. Reus and J. Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 2006. in press.

[26] 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.

[27] R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.

[28] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

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