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

\[ C \; ; \text{if} \; B \; \text{then} \; C' \; \text{else} \; C'' = \text{if} \; B \; \text{then} \; (C;C') \; \text{else} \; (C;C'') \]

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 \textit{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):

\[
x=m1(); \quad y=m2(); \quad m3(x,y);
\]
\[
y=m2(); \quad x=m1(); \quad m3(x,y);
\]

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.
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.

The semantics of types is now given as follows:

\[
\begin{align*}
\text{dom}(\emptyset) &= \emptyset \\
\text{dom}(s[l\rightarrow v]) &= \text{dom}(s) \\
(s[l\rightarrow v]).\ell &= \text{dom}(s) \\
\text{new}(s,v) &= (s',s') \Rightarrow \text{dom}(s') = \text{dom}(s) \cup \{\ell\} \\
\ell \notin \text{dom}(s) \land s'.\ell &= v
\end{align*}
\]

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 \(s, s' \in S\) then we define \(s \sim_L s' \Leftrightarrow \text{dom}(s) \supseteq L \land \text{dom}(s') \supseteq L \land \forall \ell \in L. s.\ell = s'.\ell\)

The semantics of types is now given as follows:

\[\begin{align*}
\text{[unit]} &= \{v\} \\
\text{[int]} &= \mathbb{Z} \\
\text{[ref]} &= \mathcal{L} \\
\text{[A \times B]} &= \text{[A]} \times \text{[B]} \\
\text{[A \rightarrow TB]} &= \text{[A]} \rightarrow \text{[TB]} \\
\text{[T\{A\}]} &= S \rightarrow S \times [A]
\end{align*}\]

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

\[\begin{align*}
[\Gamma \vdash V : A] : [\Gamma] \rightarrow [A] \\
[\Gamma \vdash M : T\{A\}] : [\Gamma] \rightarrow [T\{A\}]
\end{align*}\]

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, \(V_{sun}\), computes the sum of its three arguments using a reference to store an intermediate result. Here \(\text{int}\)\(^3\) abbreviates \(\text{int} \times \text{int} \times \text{int}\) 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, \(M_{sun}\), 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, \(M_{out}\), 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 \(\text{Regs}\) of region identifiers (regions for short) ranged over by \(r\). Types and typing judgments will involve only finitely many regions. Such finite sets are ranged over by \(I\). 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 \(\epsilon\), is a finite set of primitive effects. If \(\epsilon\) is an effect, we define the \emph{reads of} \(\epsilon\) by \(\text{rds}(\epsilon) = \{r \mid rd_r \in \epsilon\}\) and similarly its \emph{writes} \(\text{wrs}(\epsilon)\) and its \emph{allocations} \(\text{als}(\epsilon)\).

\(^1\)This is not quite the free implementation because of the \textit{new} operation.

### 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 clean-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 \(T\) and contexts \(\Gamma\) as follows:

\[
\begin{align*}
A, B &::= \text{unit} | \text{int} | \text{bool} | \text{ref} | A \times B | A \rightarrow TB \\
\Gamma &::= x_1 : A_1 \ldots x_n : A_n
\end{align*}
\]

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 \(\Gamma \vdash M : T\{A\}\), defined inductively by the rules in Figure 1. Note that the types on \(\lambda\)-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.
Γ ⊢ n : int \quad Γ ⊢ b : bool \quad Γ ⊢ () : unit \quad Γ, x : A ⊢ x : A

Γ ⊢ V_1 : int \quad Γ ⊢ V_2 : int

Γ ⊢ V_1 + V_2 : int

Γ ⊢ V_1 : A \quad Γ ⊢ V_2 : B

Γ ⊢ (V_1, V_2) : A \times B

Γ, x : A ⊢ M : TB

Γ ⊢ V_1 : A → TB \quad Γ ⊢ V_2 : A

Γ ⊢ V : A

Γ ⊢ M : TA \quad Γ, x : A ⊢ N : TB

Γ ⊢ V : bool \quad Γ ⊢ M : TA \quad Γ ⊢ N : TA

Γ ⊢ let x ⇥ M in N : TB

Γ ⊢ V : ref

Γ ⊢ V_1 : ref \quad V_2 : int

Γ ⊢ V : int

Γ ⊢ read(V) : Tint

Γ ⊢ write(V_1, V_2) : Tunit

Γ ⊢ ref(V) : Tref

\begin{align*}
M_1, M_2 & := \text{let } x ⇥ M_1 \text{ in } M_2 \quad (\text{sequential composition}) \\
V_{\text{sum}} & := \lambda (x, y, z) : \text{int}^3. \text{let } x ⇥ \text{ref}(x + y) \text{ in } \text{read}(x) + z : \text{int}^3 \rightarrow T\text{int} \\
M_{\text{int}} & := \text{let } x ⇥ \text{ref}(0) \text{ in } (\lambda z : \text{unit}. \text{read}(x), \lambda z : \text{unit}. \text{write}(x, \text{read}(x) + 1)) : T\text{(unit → Tint) × (unit → Tunit)}) \\
M_{\text{mem}} & := \text{let } x ⇥ \text{ref}(0) \text{ in } \text{let } y ⇥ \text{ref}(1) \text{ in } \lambda a : \text{int}. \text{if } a = \text{read}(x) \text{ then } \text{read}(y) \text{ else } \text{write}(x, a); \text{write}(y, a + 1); \text{read}(y) : T(\text{int → Tint}) \\
M_{\text{mut}} & := \text{let } x ⇥ \text{ref}(0) \text{ in } \lambda a : \text{int}. \text{let } o ⇥ \text{read}(x) \text{ in } \text{write}(x, a); o : T(\text{int → Tint})
\end{align*}

\begin{figure}[h]
\centering
\begin{align*}
\lambda & \vdash n : \text{int} \\
\lambda & \vdash b : \text{bool} \\
\lambda & \vdash () : \text{unit} \\
\lambda, x & \vdash x : A
\end{align*}
\caption{Simple computation type system}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\lambda & \vdash \text{let } x ⇥ M \text{ in } N : TB \\
\lambda & \vdash V : \text{ref} \\
\lambda & \vdash V_1 : \text{ref} \quad V_2 : \text{int}
\end{align*}
\caption{Example programs}
\end{figure}

We define \textit{refined} value types ranged over by X and Y:

\begin{align*}
X, Y & := \text{unit} \mid \text{int} \mid \text{bool} \mid \text{ref} \mid X \times Y \mid X → Y
\end{align*}

A refined computation type takes the form \( T_\varnothing X \) with \( X \) a refined value type. A refined typing context \( \Theta \) is a finite map from variables to refined value types. The well-formedness judgements \( \Pi \vdash \varepsilon \varnothing \) and \( \Pi \vdash \Theta \varnothing \) mean that only regions declared in \( \Pi \) appear in \( \varepsilon \), \( X \) and \( \Theta \). 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 \( \lambda \)-bound variables).

\begin{lemma}
If \( \Pi, \Theta \vdash V : X \) then \( U(\Theta) \vdash V : U(X) \), and similarly for computations.
\end{lemma}

The last rule in Fig. 4 is called the \textit{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., \( \Pi = \{ r \} \) and \( \varepsilon = \{ \text{rd}_r, \text{wr}_r, \text{al}_r \} \) 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.

\subsection{4.1 Example programs}

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

\begin{align*}
\lambda & \vdash \text{let } x ⇥ \text{ref}(0) \text{ in } \text{let } y ⇥ \text{ref}(1) \text{ in } \lambda a : \text{int}. \text{if } a = \text{read}(x) \text{ then } \text{read}(y) \text{ else } \text{write}(x, a); \text{write}(y, a + 1); \text{read}(y) : T(\text{int → Tint}) \\
\lambda & \vdash \text{let } x ⇥ \text{ref}(0) \text{ in } \lambda a : \text{int}. \text{let } o ⇥ \text{read}(x) \text{ in } \text{write}(x, a); o : T(\text{int → Tint})
\end{align*}

The function \( V_{\text{sum}} \) can even be typed as

\begin{align*}
\lambda & \vdash \text{let } x ⇥ \text{ref}(0) \text{ in } \lambda a : \text{int}. \text{let } o ⇥ \text{read}(x) \text{ in } \text{write}(x, a); o : T(\text{int → Tint})
\end{align*}

Thus will be (correctly) regarded as pure by our analysis. Unfortunately, the masking rule does not apply to the type of the pure computation \( M_{\text{mem}} \). Indeed, if it did, the impure \( M_{\text{mut}} \) which has the same type as \( M_{\text{mem}} \) would be falsely given a pure typing.
int a computation of an end result of type M spirit of Haskell’s “runST” \[10\] is as follows. Suppose that M T the context. For example, we have arbitrary and can thus be chosen differently from regions used in M to externally visible portions of the store (formally: the regions J where ∀). straightforwardly by interpreting polymorphic types as (possibly soundness can be extended to the effect-polymorphic case quite effects or types) for the sake of simplicity. Our proof of type refrain from considering polymorphism (at the level of regions, A schematic example for the use of the masking rule in the All these typings only use a single region which, however, is 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].

Figure 3. Subtyping refined types

| Π; Θ ⊢ n : int | Π; Θ ⊢ b : bool | Π; Θ ⊢ () : unit | Π; Θ, x : X ⊢ x : X |
| Π; Θ ⊢ V₁ : int | Π; Θ ⊢ V₂ : int |
| Π; Θ ⊢ V₁ : X | Π; Θ ⊢ V₂ : Y |
| Π; Θ ⊢ (V₁, V₂) : X × Y |
| Π; Θ ⊢ V₁ : X → Tₑ Y | Π; Θ ⊢ V₂ : X |
| Π; Θ ⊢ let x := M in N : Tₑₘₐₓ Y |
| Π; Θ ⊢ V : ref, |
| Π; Θ ⊢ read(V) : Tₑₙₑₓ(it) |
| Π; Θ ⊢ write(V₁, V₂) : Tₑₙₑₓ(unit) |
| Π; Θ ⊢ ref(V) : Tₑₙₑₓ(ref) |
| Π; Θ ⊢ V : X X ≤ X' |
| Π; Θ ⊢ M : Tₑ X Tₑ X ≤ Tₑ X' |
| Π; Θ ⊢ M : Tₑ X X ≤ ok |
| Π; Θ ⊢ M : Tₑ \{rdₑ, wrₑ, alₑ\} X |

Figure 4. Refined type system

5. Formal preliminaries

In our previous work [8] we approximated contextual equivalence by a partial equivalence relation \[\llbracket X \rrbracket\] 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 \(L\) and \(f \subseteq L \times L'\) such that \((l₁, l'₁)\) ∈ \(f\) and \((l₂, l'₂)\) ∈ \(f\) imply \(l₁ = l₂\) ⇔ \(l'₁ = l'₂\).

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

Two partial bijections \(t₁, t₂\) are disjoint if \(dom(t₁) \cap dom(t₂) = \emptyset\) and \(dom'(t₁) \cap dom'(t₂) = \emptyset\). In this case, we write \(t₁ \otimes t₂\) for
the partial bijection given by
\[ t_1 \otimes t_2 = \{ \text{dom}(t_1) \cup \text{dom}(t_2), \text{dom}^\prime(t_1) \cup \text{dom}^\prime(t_2), t_1 \cup t_2 \} \]
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 \( \tau \not\in \text{Regs} \) 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 \( \tau \) as the silent region. This intended meaning will become clear subsequently; for now, \( \tau \) is just a symbol.

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

Definition 2 (parameter). A parameter \( \varphi \) is a pair \((\Pi, f)\) where
- \( \Pi \) is a finite set of regions,
- \( f \) is a function assigning to each region \( r \in \Pi \cup \{\tau\} \) a partial bijection \( f(r) \) 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:
- \( \text{reg}(\varphi) = \Pi \)
- \( \varphi(r) = \{ f(r) \) when defined \}
- \( \varphi(r) = \emptyset \) otherwise
- \( \text{dom}(\varphi) = \bigcup_{r \in \text{reg}(\varphi)} \text{dom}(\varphi)(r) \)
- \( \text{dom}^\prime(\varphi) = \bigcup_{r \in \text{reg}(\varphi)} \text{dom}^\prime(\varphi)(r) \)

Note that \( \text{dom}(\varphi) \) and \( \text{dom}^\prime(\varphi) \) are both finite. We define \( \text{Par}(\Pi) = \{ \varphi \mid \text{reg}(\varphi) \subseteq \Pi \} \). We also use the notation \( \Pi \vdash \varphi \) ok to mean \( \varphi \in \text{Par}(\Pi) \). If \( \varphi \) and \( \varphi' \) are parameters such that
\( \text{dom}(\varphi) \cap \text{dom}(\varphi') = \text{dom}^\prime(\varphi) \cap \text{dom}^\prime(\varphi') = \emptyset \)
then \( \varphi, \varphi' \) are called disjoint and we write \( \varphi \otimes \varphi' \) for the obvious juxtaposition of \( \varphi \) and \( \varphi' \). Formally,

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

Whenever we write \( \varphi \otimes \psi \) then \( \varphi \) and \( \psi \) are presumed to be disjoint from each other so, a statement like \( \exists \psi_1 \ldots \varphi \otimes \psi \ldots \) is understood as “there exists \psi disjoint from \varphi such that \ldots \varphi \otimes \psi \ldots”.

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

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

Thus, if \( t \not\in \text{dom}(\varphi) \) and \( t' \not\in \text{dom}^\prime(\varphi) \) then we can form \( \varphi \otimes [r \mapsto t'] \) to add the link \( (t, t') \) to \( r \) in \( \psi \). Similarly, if \( t \not\in \Pi \), we can form \( \varphi \otimes [r \mapsto \tau] \) to initialise a new region \( r \) with \( \emptyset \).

If \( \Pi, r \vdash \varphi \) ok then \( \varphi \mapsto r \) denotes the parameter such that \( \Pi \vdash \varphi \mapsto r \) ok and
\( \varphi(r') = \varphi(t') \text{ when } r' \neq r \)
(\( \varphi(\tau) = \varphi(\tau) \otimes (\text{dom}(\varphi)(\tau), \text{dom}^\prime(\varphi)(\tau), \emptyset) \))

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 \( R \subseteq S \times S \) 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 \( \text{StRel}(L, L') \) for the set of all store relations on \( L, L' \).

Given such a relation, we now formalize what it means to ‘respect’ an effect \( e \) under some parameter \( \varphi \).

Definition 4 (relations and effects). Suppose \( \Pi \vdash \varphi \) ok and \( \Pi \vdash e \) ok. Let \( R \) be a store relation on \( \text{dom}(\varphi), \text{dom}^\prime(\varphi) \). We say that \( R \) respects \( e \) at \( \varphi \) if it is preserved by all commands that exhibit only \( e \) on the store layout delineated by \( \varphi \). Formally, we define:
- \( R \) respects \( \{vd\} \) at \( \varphi \) if \( s, s' \in R \) implies \( s, s' \in R \) for all \( (t, t') \in \varphi(e) \).
- \( R \) respects \( \{ur\} \) at \( \varphi \) if \( s, s' \in R \) and for all \( (t, t') \in \varphi(e) \) and \( v \in \mathbb{Z} \), we have \( (s[t\mapsto v], s'[t'\mapsto v]) \in R \);
- \( R \) respects \( \{al\} \) always.

We then define the set \( \text{R}^\Pi(\varphi) \) of all store relations that respect \( e \) at \( \varphi \) as follows:
\( \text{R}^\Pi(\varphi) = \{ R \in \text{StRel}(\text{dom}(\varphi), \text{dom}^\prime(\varphi)) \mid \forall c : e, R \text{ resp. } c \text{ at } \varphi \} \).

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 \( s, s' \in S \) we define
\( s, s' \models \varphi \iff \text{dom}(s) = \text{dom}(\varphi) \land \text{dom}(s') = \text{dom}^\prime(\varphi) \)
We also define the following:
\( s \sim_{\varphi} s' \iff \forall r \in \text{Regs}. (\forall (t, t') \in \varphi(e), s, t = s', t' = t') \).

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 \( \llbracket X \rrbracket^\Pi \subseteq \llbracket U(X) \rrbracket \times \llbracket U(X) \rrbracket \) by
\( \llbracket X \rrbracket^\Pi = \Delta_{\llbracket U(X) \rrbracket} \) when \( X \in \{ \text{int}, \text{bool}, \text{unit} \} \)
\( \llbracket \text{ref}_X^\Pi \rrbracket = \varphi(r) \)
\( \llbracket X \times Y \rrbracket^\Pi = \llbracket X \rrbracket^\Pi \times \llbracket Y \rrbracket^\Pi \)
\( \llbracket X \rightarrow T, Y \rrbracket^\Pi = \{ (f, f') \mid \forall \varphi \geq_{\Pi} \varphi, \forall (x, x') \in \llbracket X \rrbracket^\Pi, (f(x), f'(x')) \in \llbracket T, Y \rrbracket^\Pi \}
\( \llbracket T, X \rrbracket^\Pi = \{ (f, f') \mid s, s' \models \varphi \Rightarrow \forall R \in \mathbb{R}^\Pi(\varphi), s, s' \Rightarrow s_1 R s_1' \land \exists \psi \in \text{Par}(\text{als}(e)), s_1, s_1' \models \varphi \otimes \psi \land s_1 \sim \psi s_1' \land (v, v') \in \llbracket X \rrbracket^\Pi \}
\)
We define \( \llbracket \Theta \rrbracket^\Pi \) by \( \llbracket \Theta \rrbracket^\Pi = \{ (\gamma, \gamma') \mid \forall i. (\gamma(x_i), \gamma'(x_i)) \in \llbracket X_i \rrbracket^\Pi \} \) where \( \Theta = \{ x_i: X_i, \ldots, x_n: X_n \} \).

The definition of the logical relation on computation types deserves some explanation. First, it says that the store behaviour of
Lemma 2 (Monotonicity). Suppose \( \Pi \vdash X \) ok. If \( \varphi' \geq_{\Pi} \varphi \) then \( [X]_{\varphi'} \supseteq [X]_{\varphi} \).

Proof. By induction on \( X \). □

Lemma 3 (Masking). Suppose \( \Pi \vdash X \) ok and \( \Pi, r \vdash \varphi \) ok. Then \( [X]_{\varphi'r} = [X]_{\varphi} \) and likewise \( [T_r(X)]_{\varphi'r} = [T_r(X)]_{\varphi} \).

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 \( X = \text{ref}_r \), then \( r' \neq r \).

Case \( X = X_1 \to T_r X_2 \). Suppose that \( (f, f') \in [X_1]_{\varphi'} \) and that \( \varphi' \geq_{\Pi} \varphi \) and that \( (x, x') \in [X_1]_{\varphi} \). Write \( \varphi' = (\varphi - r) \otimes \theta \) and put \( \psi = \varphi \otimes \theta \). We have \( \psi \geq_{\Pi} \varphi \) and \( (x, x') \in [X_1]_{\psi} \), the induction hypothesis since \( \psi - r = \varphi' \). Note that \( \divides_{\Pi} \otimes \varphi \) and so \( \Pi \vdash \varphi \). The assumption then gives \( (f, f', v') \in [T_r X_2]_{\varphi'} \) and hence \( (f, f' v') \in [T_r X_2]_{\psi} \) as required, again by the induction hypothesis and \( \psi - r = \varphi' \).

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

Case \( T_r(X) \). Suppose that \( (f, f') \in [T_r(X)]_{\varphi'} \) and \( \varphi 

Lemma 4 (Extension). \( R^\Pi_{\varphi'}(\varphi) = R^\Pi_{\varphi}(\varphi \otimes [r \to \emptyset]) \).

Proof. The \( \otimes \) direction is obvious. Conversely, if \( R \) respects \( \varepsilon - r \) at \( \varphi \) then it also respects \( \varepsilon \) at \( \varphi \otimes [r \to \emptyset] \) since the additional restrictions imposed by possible occurrences of \( r \) in \( \varepsilon \) are vacuously true at \( \varphi \otimes [r \to \emptyset] \).

The following establishes semantic soundness for our subtyping relation.

Lemma 5 (Soundness of subtyping). If \( \Pi \vdash X_1 \) ok and \( X_1 \subseteq X_2 \) then for all \( \varphi \in \divides_{\Pi} \) one has \( [X_1]_{\varphi'} \subseteq [X_2]_{\varphi} \).

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, \( \Pi \vdash V : X \) and \( (\gamma, \gamma') \in [\Theta]_{\varphi} \), then
   \( ([U(\Theta) \vdash V : U(X)]_{\gamma,\gamma'} \subseteq [X]_{\varphi} \).

2. If \( \Pi \vdash \varphi \) ok, \( \Pi \vdash M : T(U(X)) \) and \( (\gamma, \gamma') \in [\Theta]_{\varphi} \), then
   \( ([U(\Theta) \vdash M : T(U(X))]_{\gamma,\gamma'} \subseteq [T_r(X)]_{\varphi} \).

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

Case 1a. From the derivation we have \( \Pi, \Theta \vdash M_1 : T_{\kappa}(X_1) \).

Case 2a. Assume \( \Pi \vdash \varphi \) ok and \( (\gamma, \gamma') \in [\Theta]_{\varphi} \). Suppose \( R \in \mathcal{R}_{\varphi}^{\Pi} \) and \( s_0 R s_0 \). Let

\[ f_1 = [U(\Theta) \vdash M_1 : T U X_1], \]

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

\[ f_2 = [U(\Theta) ; x : U(X_1) \vdash M_2 : T U X_2], \]

and \( (s_2, v_2) = f_2(\gamma, v_1)s_1 \) and \( (s_2', v_2') = f_2(\gamma', v_1')s_1' \).

Now \( R \in \mathcal{R}_{\varphi}^{\Pi} \) holds for \( i = 1, 2 \). The induction hypothesis for (1), applied to \( \varphi, (\gamma, \gamma') \) and \( (s_0, s_0') \), therefore yields \( s_1 R s_1' \) and also furnishes \( s_1 \in \divides_{\Pi} (s_1, s_1') \) such that \( s_1, s_1' \divides_{\Pi} \varphi \) and \( (s_1, s_1') \in [X]_{\varphi} \) where \( \varphi' = \varphi \otimes \psi \). We therefore define a relation \( R^* \)

\[ s R^* s' \iff \exists s R s' \land s \sim_{\varphi} s' \]

We then have \( R \in \mathcal{R}_{\varphi}^{\Pi} \) and also \( s_1 R^* s_1' \). In particular, \( R^* \) respects \( rd \), \( r \) in view of the clause \( s \sim_{\varphi} s' \) that has been added for that purpose.

The monotonicity of the interpretation of the value types (Lemma 2) yields \( (\gamma(\gamma, \gamma'), (\gamma', \gamma')) \in [\Theta, X]_{\varphi}^{\Pi} \). We can therefore apply the induction hypothesis for (2), with \( \varphi', \gamma(\gamma, \gamma'), (\gamma', \gamma') \) and \( (s_1, s_1') \), and obtain the existence of some \( s_2 \in \divides_{\Pi} (s_2, s_2') \) such that \( s_2, s_2' \divides_{\Pi} \varphi \otimes \psi \) and \( s_2 R^* s_2' \) and \( s_2 \sim_{\varphi} s_2' \). From the definition of \( R^* \) we then get \( s_2 R^* s_2' \) and \( s_2 \sim_{\varphi} s_2' \).
Now, since $R$ is a store relation on $\text{dom}(\phi), \text{dom}'(\phi)$ and $s_0, s'_0 \models \phi$ we find $s_1R\gamma_i$. We let $w = [n-\ell, \ell', \ell]''$ and have $s_1, s'_1 \models \phi \otimes \psi$. We have $[\ell, \ell'] \in \ref \Pi$ and since $[U\Theta + V : \text{int}]\gamma = [U\Theta + V : \text{int}]\gamma'$ from $(\gamma, \gamma') \in [\Theta]^{\Pi}$, we also have $s_1 \sim \psi s'_1$ and we are done.

**Masking Rule** Suppose $\Pi \vdash \theta ok, \Pi \vdash X ok, \Pi, r; \Theta \vdash M : T_i X, \Pi \vdash \varphi ok$ and $R \in \mathcal{R}^{\theta, \varphi}_i(\psi)$ and $s_0R\gamma_0$ and $s_0, s_0 \models \phi$. Write $g := [U\Theta + M : T_i U X]$ and $(s_1, v_1) = g\gamma_0$ and $(s'_1, v'_1) = g\gamma_0'$. Let us write $\psi = \varphi \otimes [n-\theta]$. The masking lemma 3 shows $(\gamma, \gamma') \in [\Theta]^{\Pi, r}$ and from the extension lemma 4 we have $R \in \mathcal{R}^{\theta, \varphi}_i(\psi)$. So the induction hypothesis furnishes $\theta \in \text{Par}(\phi)$ and disjoint from $s_0, s'_0$ such that $s_1, s'_1 \models \varphi \psi$ and $s_1 \sim s'_1$ and $s_1R\gamma_i$ and $(v_1, v'_1) \in [X]^{\Pi, r}$ where $\psi' = \psi \otimes \theta$. Now, $\psi' \sim r \varphi \otimes (\theta-r)$ and so $(v_1, v'_1) \in [X]^{\Pi, r}(\theta-r)$ again by the masking lemma. The rest is clear from the definitions.

\[\square\]

**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 $\Pi; \emptyset \vdash V : X \rightarrow T_i \text{unit} \langle \langle \text{"context"} \rangle \rangle$ with $\varepsilon$ 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 $\Pi; \emptyset \vdash M_1 \equiv M_2 : A$ 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., $X = (\text{int} \rightarrow T_i \text{unit}) \rightarrow T_i \text{unit}$ and the equivalence of $V_1$ and $V_2$ relies on their being fed a pure function as input. E.g., in such a system 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 $\Pi; \Theta \vdash M_i : T_i X$ for $i = 1, 2$ and whenever $\Pi \vdash \varphi ok$ and $(\gamma, \gamma') \in [\Theta]^{\Pi}$ then 

\[(U(\Theta) + M_1 : T(U(X))\gamma), (U(\Theta) + M_2 : T(U(X))\gamma') \in [T_i X]^{\Pi}].\]

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 $\Theta \vdash T_i A$ for the common type of these values. It is easy to see from the definition of the logical relation that 

\[((\lambda M_1)), (\lambda M_2)) \in \mathcal{R}(\Theta = T_i X)]^\Pi\]

for all $\varphi$ (we have omitted types and contexts here). Now let $V : (\Theta \rightarrow T_i X) \rightarrow T_i \text{bool} \langle \langle \text{"context"} \rangle \rangle$. By applying the fundamental lemma to $V$ we conclude 

\[((V \lambda M_1)), (V \lambda M_2)) \in [T_i \text{bool}]^{\Pi}\]

Now, let $\varphi$ 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 $R \in \mathcal{R}_{\emptyset, \emptyset} (\varphi)$ be arbitrary. Clearly, $\emptyset, \emptyset \models \varphi$ and $\varphi' \models \text{unit}$. It expresses that $\varphi$ and $\varphi'$ agree on those locations that are read given effect $\varepsilon$.

We also define 

\[\text{wrs}(\varphi) = \text{dom}(\varphi) \setminus \text{wrs}(\varphi) \text{dom}(\varphi) \text{dom}(\varphi) \setminus \text{wrs}(\text{dom}(\varphi))\]

Then $\text{wrs}(\varphi)$ and $\text{wrs}(\varphi')$ comprise the locations on the left (resp. right) side that are not written to given effect $\varepsilon$. This includes the locations in the silent region.

**Lemma 6.** Suppose $\Pi; \Theta \vdash M : T_i X$ and $\Pi \vdash \varphi ok$ and $(\gamma, \gamma') \in [\Theta]^{\Pi}$ and $s_0, s'_0 \models \varphi$ and $[M]_\gamma g_0 = (s_1, x')$.

If $s_0 \sim_{\text{red}_{\text{wrs}}(\varphi)} s'_0$ then there exists $\psi \in \text{Par}(\phi)$ 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 $s_1 \sim s'_1$.

2. $s_0 \sim_{\text{wrs}(\varphi)} s_1$ and $s'_0 \sim_{\text{wrs}(\varphi)} s_1'$.

3. For each $(\ell, \ell') \in \varphi$ where $r \in \Pi$ we have either

- $s_0, \ell = s_1, \ell$ and $s'_0, \ell' = s'_1, \ell'$ (unchanged) or
- $s_0, \ell = s'_0, \ell'$ (identically written).

4. Suppose that $\ell \in \text{dom}(\varphi)$ but there is no $\ell'$, $r$ such that $(\ell, \ell') \in \varphi$. 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 

\[(s, s') \in R \iff s \sim_{\text{red}_{\text{wrs}}(\varphi)} s' \land (s, s') \in \text{wrs}(\varphi) \land s_0 \sim s_0'.\]

We now have $s_0R_0$ (not necessarily $s_0R'_0$) and so $s_1R_1$. The claim about $s_1$ follows. The proof for $s_0 \sim s_0'$ is analogous.

For part 3 we define 

\[(s, s') \in R \iff s \sim_{\text{red}_{\text{wrs}}(\varphi)} s' \land (s, s') \in (s, s'), s_0, s'_0 \sim s_0'.\]

and note that $R \in \mathcal{R}_{\emptyset, \emptyset} (\varphi)$. It follows $s_1, s'_1 \in R$ and the claim follows.

Part 4, finally, follows using the relation 

\[sRs' \iff s \sim_{\text{red}_{\text{wrs}}(\varphi)} s' \land s, s' \in R.\]

\[\square\]

**Proposition 2 (duplicated computation).** Suppose that $\Pi; \Theta \vdash M : T_i X$ and suppose that $\text{rds}(\varepsilon) \cap \text{wrs}(\varepsilon) = \text{als}(\varepsilon) = \emptyset$. Thus,
Suppose that $\psi \in \text{Par}(\emptyset)$ disjoint from $s_0, s_0'$ such that $s_2, s_2' \models \varphi$ and $(v_2, v_2') \in [X]_{\psi''} \cup [V]_{\psi''}$. Let $v, v' \in s$ be such that $v = s_0$ and $v' = s_0'$. Then, $\psi$ is closed under the right-hand side, we find that $\psi_1$ and $\psi_2$ are disjoint allowing us to put $\psi := \psi_1 \uplus \psi_2$ and $(v_1, v_1'), (v_2, v_2') \in [X]_{\psi''} \cup [V]_{\psi''}$ by monotonicity. We also have $s_2, s_2' \models \varphi$.

Since $s_0R_{\delta} s_0'$ we can show $s_2R_{\delta} s_2'$ by induction on the size of $\text{dom}_{\psi''}(\varphi)$ using the definition of $R_{\delta}^{\psi''}(\varphi)$.

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

$\Pi; \emptyset \vdash M : T_e(\text{unit})$

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

**Proof.** Let $\varphi \in \text{Par}(\Pi)$ and $(\gamma, \gamma') \in [\Theta]_\varphi$. Let $f = [M]_\gamma$ and suppose $s_0, s_0' \models \varphi$. Let $v, v_1, f = s_0'$. Of course, $v = \ast$. The fundamental lemma applied to $M$, $\varphi$, $s_0, s_0'$ furnishes $\psi \in \text{Par}(\ast)$ disjoint from $\varphi$. Construct $\psi$ by taking only the left part of $\psi$; formally: $v_1(r) = \{\text{dom}(\psi)(r), \emptyset, \emptyset\}$. Now, $s_1, s_0' = \varphi \equiv \psi_1$ and, trivially, $(v, v_1) \in [\text{unit}]_{\psi_1}$. If $R \in R_{\delta}^{\psi''}(\varphi)$ and $s_0R_{\delta} s_0'$ then $s_1$ agrees with $s_0$ on all locations that $R$ depends upon, so $s_1R_{\delta} s_0'$.

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

$\Pi; \emptyset \vdash M : T_e(\text{unit})$

**and that $M \in T_e(X \times X)$.**

**Put**

$M_1 := \text{val}(\lambda x : X. y \mapsto M \in N)$

$M_2 := \text{let } y \mapsto M \text{ in } \text{val}(\lambda x : X. N)$

**Then**

$\Pi; \emptyset \vdash M_1 \equiv M_2 : T_e(X \times X)$

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

$f = [M]_\gamma$

$f' = [M]_{\gamma'}$

$g(v_1, v_2) = [N]_\gamma[x \mapsto v_1, y \mapsto v_2]$

$g'(v_1, v_2) = [N]_{\gamma'}[x \mapsto v_1, y \mapsto v_2]$

Suppose that $s_0, s_0' \models \varphi$. The fundamental theorem applied to $M$ yields $\psi \in \text{Par}(\emptyset)$ such that $s_1, s_1' \models \varphi \equiv \psi$ where $s_1, s_1' \models \varphi \equiv \psi$. We claim that

$\lambda x, y.g(v_1, v_2), \lambda x, y.s_1', v_2'. s_2, v_2'. s_2' \models \varphi$.

Assume thus $\varphi \equiv \psi \equiv \varphi'$ and $(v_2, v_2') \in [X]_{\varphi}$. Also assume $s, s' \models \varphi$ and $s_0, s_1' \models \varphi$. We would like $\lambda x, y.g(v_1, v_2), s_1', v_2'. s_2, v_2'. s_2' \models \varphi'$. Now, $s_0, s_0' \models \varphi$.
Examples Suppose that the let bindings \( f_1 \equiv M_{\text{init}} \) and \( f_2 \equiv M_{\text{init}} \) 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

\[
\begin{align*}
\text{let } x & \equiv f_1(5) \text{ in } \text{let } y \equiv f_2(4) \text{ in } \text{let } z \equiv f_2(6) \text{ in } (x, z) \\
\text{let } x & \equiv \begin{pmatrix} f_1(5) \end{pmatrix} \text{ in } \text{let } y \equiv \begin{pmatrix} f_2(4) \end{pmatrix} \text{ in } \text{let } z \equiv \begin{pmatrix} f_2(6) \end{pmatrix} \text{ in } (x, z)
\end{align*}
\]

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

\[
\lambda x. \text{let } s \equiv V_{\text{assg}}(a, b, c) \text{ in } \text{val } (x + s)
\]

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 \( T_0(A) \) 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 type system for a higher-order language with dynamically allocated store and higher order features. 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 calculus 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.⁴

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 \( [T, X]^T \) with the least admissible relation comprising it. This is because the existential quantification in the definition of this relation thwart’s an attempt to show admissibility by induction on types. The least admissible relation \( R' \) 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 \( dRd' \) and one wants to show some property \( \psi(d, d') \) where \( \psi \) is admissible then one can w.l.o.g. assume that one has \( dd' \).

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 \( M_1 \equiv M_2 \) 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 \( (x=M_1; y=M_1; z,x) \equiv (x=M_1; (z,x)) \), however, we have to prove it under the weaker assumption \( M \equiv M \). This is an example of the difference between proving particular and generic equivalences.

³The more general relationship between logical relations and bisimulation still seems slightly murky (at least to us) and clearly demands further study as well.

⁴In fact, rather more holds: the continuation-based formulation uses a ‘\( \top \) 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

$$M_{\text{mem}} : T_0 (\text{int} \rightarrow T_0 \text{int})$$

for $M_{\text{mem}}$ and similar memoised functions on the grounds that they behave just as a pure function. With such a typing, an instance of $M_{\text{mem}}$ could be hoisted out of an abstraction which it presently cannot. One way to make our semantics believe that $M_{\text{mem}}$ 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 $M_{\text{mem}}$, one could use

$$\text{read}(y) = \text{read}(x) + 1$$

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 $M_{\text{mem}}$ as pure yet of course reject $M_{\text{read}}$ as impure.

**General references** Of course it would be interesting to include references to types other than integers. References to product and general references to types other than integers. References to product and similar memoised functions on the grounds that they behave just as a pure function. With such a typing, an instance of $M_{\text{mem}}$ could be hoisted out of an abstraction which it presently cannot. One way to make our semantics believe that $M_{\text{mem}}$ 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 $M_{\text{mem}}$, one could use

$$\text{read}(y) = \text{read}(x) + 1$$

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 $M_{\text{mem}}$ as pure yet of course reject $M_{\text{read}}$ as impure.

**References**


