Reading, Writing and Relations
Towards Extensional Semantics for Effect Analyses

Nick Benton[1], Andrew Kennedy[1], Martin Hofmann[2], and Lennart Beringer[2]

[1] Microsoft Research, Cambridge
[2] Ludwig-Maximilians-Universität, München

Abstract. We give an elementary semantics to an effect system, tracking read and write effects by using relations over a standard extensional semantics for the original language. The semantics establishes the soundness of both the analysis and its use in effect-based program transformations.

1 Introduction

Many analyses and logics for imperative programs are concerned with establishing whether particular mutable variables (or references or heap cells or regions) 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.[1]

Effect systems, first introduced by Gifford and Lucassen [8,11], 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 no truly satisfactory account 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 [3,6,4], 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

[1] Though here we restrict attention, in an essential manner, to sequential programs.

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 the case of not writing a variable, a naive extensional interpretation seems clear: a command C *does not observably write the variable* X if it leaves the value of X unchanged:

$$\forall S, S'.\ C, S \Downarrow S' \implies S'(X) = S(X)$$

Note that this definition places no constraint on diverging executions or the value of X at intermediate states. Operationally, C may read and write X many times, so long as it always restores the original value before terminating. Furthermore, the definition is clearly closed under behavioural equivalence. If we have no non-termination and just two integer variables, X and Y, and the denotation of C is $[\![C]\!] : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ then our simple-minded definition of what it means for C not to write X can be expressed denotationally as

$$\exists f_2 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}. \forall X, Y.\ [\![C]\!](X, Y) = (X, f_2(X, Y))$$

which is the same as saying $[\![C]\!] = \langle \pi_1, f_2 \rangle$.

The property of *neither reading nor writing* X, i.e. of being *observationally pure in* X is also not hard to formalize extensionally:

$$\forall S, S', n.\ C, S \Downarrow S' \iff C, S[X \mapsto n] \Downarrow S'[X \mapsto n]$$

Alternatively $\exists f_2 : \mathbb{Z} \to \mathbb{Z}. \forall X, Y.\ [\![C]\!](X, Y) = (X, f_2(Y))$, which is the same as saying $[\![C]\!] = 1 \times f_2$.

The property of *not observably reading* X is rather more subtle, since X may, or may not, be written. We want to say that the final values of all the other variables are independent of the initial value of X, but the final value of X itself is either a function of the other variables or is the initial value of X:

$$\exists f_1 : \mathbb{Z} \to \mathbb{B}, f_2, f_3 : \mathbb{Z} \to \mathbb{Z}. \forall X, Y.\ [\![C]\!](X, Y) = (f_1(Y) \supset X \mid f_2(Y), f_3(Y))$$

This is clearly a more complex property than the others. Another way to think of it is that the final values of the variables other than X are functions of the initial values of those variables and that for each value of those other variables, the (curried) function mapping the initial value of X to its final value is either constant or the identity. The tricky nature of the 'does not read' property also shows up if one tries to define a family of monads in a synthetic, rather than an analytic fashion (as in Tolmach's work [20]): neither reading nor writing corresponds to the identity monad; not writing corresponds to the reader (environment) monad; but there is no *simple* definition of a 'writer' monad.

Our basic approach to the soundness of static analyses and optimizing transformations is to interpret the 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. We have previously [3] described how such an extensional relational interpretation of static analyses allows one both to express constancy and dependency properties for simple imperative programs, and to reason about the transformations they enable. But the non-parametric relations used in that work turn out to be insufficient to admit a compositional, generic translation of perhaps the simplest static analysis there is: the obvious inductive definition of possibly-read and possibly-written variable sets for while-programs.

In earlier, operationally-based, work [6] we expressed the meaning of some simple global (i.e. treating the whole store monolithically) effects using sets of cotermination tests (pairs of contexts) written explicitly in the language, but those definitions were *very* unwieldy and phrased in a way that would not generalize easily to other types. Here we will show how reading, writing and allocating properties for a higher-order language with state can be elegantly captured using parametric logical relations over a simple denotational semantics for the original language. This new interpretation of effects is dramatically slicker and more compelling than previous ones.

1.1 Relations

We just recall some basic facts and notation. A (binary) relation R on a set A is a subset of $A \times A$. If R is a relation on A and Q a relation on B, then we define relations on Cartesian products and function spaces by

$$\begin{aligned} R \times Q &= \{((a,b),(a',b')) \in (A \times B) \times (A \times B) \;\mid (a,a') \in R,\, (b,b') \in Q\} \\ R \to Q &= \{(f,f') \in (A \to B) \times (A \to B) \;\mid \forall (a,a') \in R.\, (f\,a,\, f'\,a') \in Q\} \end{aligned}$$

A binary relation on a set is a *partial equivalence relation* (PER) if it is symmetric and transitive. The set of PERs on a set is closed under arbitrary intersections and disjoint unions. If R and Q are PERs, so are $R \to Q$ and $R \times Q$. Write Δ_A for the diagonal relation $\{(a,a) \mid a \in A\}$, and $a : R$ for $(a,a) \in R$.

1.2 The Basic Idea

Our starting point is the following simple, yet striking, observation, which seems not to have been made before:

Lemma 1. *For total commands operating on two integer variables, as above,*

1. *The property of not observably writing X is equivalent to*

$$\forall R \subseteq \Delta.\; [\![C]\!] : R \times \Delta \to R \times \Delta$$

i.e. preserving all relations less than or equal to the identity on X.

2. *The property of neither observably reading nor observably writing X is equivalent to*

$$\forall R.\; [\![C]\!] : R \times \Delta \to R \times \Delta$$

i.e. preserving all relations on X.

3. The property of not reading X is equivalent to

$$\forall R \supseteq \Delta.\ [\![C]\!] : R \times \Delta \to R \times \Delta$$

i.e. preserving all relations greater than or equal to the identity on X.

Proof. We write f for $[\![C]\!]$ and just consider the last case. Assume $f(X,Y) = (f_1(Y) \implies X \mid f_2(Y), f_3(Y))$ and $R \supseteq \Delta$. Then if $(X,X') \in R$ and $(Y,Y') \in \Delta$ so $Y = Y'$, we have

$$\begin{aligned}(f(X,Y), f(X',Y')) = ((&f_1(Y) \implies X \mid f_2(Y), f_3(Y)),\\ (&f_1(Y) \implies X' \mid f_2(Y), f_3(Y)))\end{aligned}$$

Clearly $(f_3(Y), f_3(Y)) \in \Delta$. In the first component, if $f_1(Y) = \mathit{true}$ then we get $(X,X') \in R$ and if $f_1(Y) = \mathit{false}$ we get $(f_2(Y), f_2(Y)) \in \Delta \subseteq R$ so we're done.

Going the other way, preservation of $\mathbb{T} \times \Delta$ deals with the independence of the second component. In the first component we need to show that for each Y, $\pi_1 f(-,Y) : \mathbb{Z} \to \mathbb{Z}$ is uniformly either constant or the identity. Pick any two distinct elements X,X' and let $R = \Delta \cup \{(X,X')\}$, which contains Δ and is therefore preserved by the first component of f. Thus $(\pi_1 f(X,Y), \pi_1 f(X',Y)) \in R$ means either $\pi_1 f(X,Y) = \pi_1 f(X',Y)$ or $\pi_1 f(X,Y) = X$ and $\pi_1 f(X',Y) = X'$. $\square$

(Note that preservation of relations is closed under unions, so it actually suffices to consider singleton relations in the first two cases and singleton extensions of the identity relation in the last case.)

In the next section we develop the result above to give a semantics for a simple effect system for a higher-order language with global variables, in the process explaining where the faintly mysterious bounded quantification really 'comes from'. The language and effect system is purposefully kept very minimal, so we may explore the key idea without getting bogged down in too much auxiliary detail.

2 Effects for Global Store

2.1 Base Language

We consider a monadically-typed, normalizing, call-by-value lambda calculus with a collection of global integer references. 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' [5], 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' [21].

We assume a finite set $\mathcal{L}$ of global variable names, ranged over by ℓ, and define value types A, computation types TA and contexts Γ as follows:

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash \texttt{read}(\ell) : T\texttt{int}} \qquad \frac{\Gamma \vdash V : \texttt{int}}{\Gamma \vdash \texttt{write}(\ell, V) : T\texttt{unit}}$$

Fig. 1. Simple computation type system

Note that 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 presence of types on lambda-bound variables makes typing derivations unique and that addition and comparison should be considered just representative primitive operations.

Since our simple language has no recursion, we can give it an elementary denotational semantics in the category of sets and functions. Writing S for $\mathcal{L} \to \mathbb{Z}$, the semantics of types is as follows:

$$[\![\texttt{unit}]\!] = 1 \qquad [\![\texttt{int}]\!] = \mathbb{Z} \qquad [\![\texttt{bool}]\!] = \mathbb{B} \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A \to TB]\!] = [\![A]\!] \to [\![TB]\!] \qquad [\![TA]\!] = S \to S \times [\![A]\!]$$

The interpretation of the computation type constructor is the usual state monad. The meaning of contexts is given by $[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$, and we can then give the semantics of judgements

$$[\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!] \qquad \text{and} \qquad [\![\Gamma \vdash M : TA]\!] : [\![\Gamma]\!] \to [\![TA]\!]$$

inductively, though we omit the completely standard details here. The semantics is adequate for the obvious operational semantics and ground contextual equivalence (observing, say, the final boolean value produced by a closed program).

2.2 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

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

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

Fig. 2. Subtyping refined types

whether a computation may read or write particular locations. Formally, define *refined* value types X, computation types $T_\varepsilon X$ and contexts Θ by

$$\begin{aligned} X, Y &:= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid X \times Y \mid X \to T_\varepsilon Y \\ \varepsilon &\subseteq \bigcup_{\ell \in \mathcal{L}} \{\texttt{r}_\ell, \texttt{w}_\ell\} \\ \Theta &:= x_1 : X_1, \ldots, x_n : X_n \end{aligned}$$

There is a subtyping relation on refined types, axiomatised in Figure 2. The evident erasure map, $U(\cdot)$, takes refined types to simple types (and contexts) by forgetting the effect annotations:

$$\begin{gathered} U(\texttt{int}) = \texttt{int} \qquad U(\texttt{bool}) = \texttt{bool} \qquad U(\texttt{unit}) = \texttt{unit} \\ U(X \times Y) = U(X) \times U(Y) \\ U(X \to T_\varepsilon Y) = U(X) \to U(T_\varepsilon Y) \\ U(T_\varepsilon X) = T(U(X)) \end{gathered}$$

$$U(x_1 : X_1, \ldots, x_n : X_n) = x_1 : U(X_1), \ldots, x_n : U(X_n)$$

Lemma 2. *If $X \leq Y$ then $U(X) = U(Y)$, and similarly for computations.* □

The refined type assignment system is shown in Figure 3. Note that the subject terms are the same (we still only have simple types on λ-bound variables).

Lemma 3. *If $\Theta \vdash V : X$ then $U(\Theta) \vdash V : U(X)$, and similarly for computations.* □

Note that the refined system doesn't rule out any terms from the original language. Define a map $G(\cdot)$ from simple types to refined types that adds the 'top' annotation $\bigcup_{\ell \in \mathcal{L}} \{\texttt{r}_\ell, \texttt{w}_\ell\}$ to all computation types, and then

Lemma 4. *If $\Gamma \vdash V : A$ then $G(\Gamma) \vdash V : G(A)$ and similarly for computations.* □

2.3 Semantics of Effects

The meanings of simple types are just sets, out of which we now carve the meanings of refined types as subsets, *together* with a coarser notion of equality.

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

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

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

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

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

$$\frac{}{\Theta \vdash \mathtt{read}(\ell) : T_{\{\mathtt{r}_\ell\}}(\mathtt{int})} \qquad \frac{\Theta \vdash V : \mathtt{int}}{\Theta \vdash \mathtt{write}(\ell, V) : T_{\{\mathtt{w}_\ell\}}(\mathtt{unit})}$$

$$\frac{\Theta \vdash V : X \quad X \leq X'}{\Theta \vdash V : X'} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad T_\varepsilon X \leq T_{\varepsilon'} X'}{\Theta \vdash M : T_{\varepsilon'} X'}$$

Fig. 3. Refined type system

More formally, the semantics of each refined type is a partial equivalence relation on the semantics of its erasure, defined as follows:

$$\begin{gathered}
[\![X]\!] \subseteq [\![U(X)]\!] \times [\![U(X)]\!] \\
[\![\mathtt{int}]\!] = \Delta_{\mathbb{Z}} \qquad [\![\mathtt{bool}]\!] = \Delta_{\mathbb{B}} \qquad [\![\mathtt{unit}]\!] = \Delta_1 \\
[\![X \times Y]\!] = [\![X]\!] \times [\![Y]\!] \\
[\![X \to T_\varepsilon Y]\!] = [\![X]\!] \to [\![T_\varepsilon Y]\!] \\
[\![T_\varepsilon X]\!] = \textstyle\bigcap_{R \in \mathcal{R}_\varepsilon} R \to R \times [\![X]\!]
\end{gathered}$$

where $\mathcal{R}_\varepsilon \subseteq \mathbb{P}(S \times S)$ is given by $\mathcal{R}_\varepsilon = \bigcap_{e \in \varepsilon} \mathcal{R}_e$ and for atomic effects e, $\mathcal{R}_e \subseteq \mathbb{P}(S \times S)$ is given by

$$\begin{aligned}
\mathcal{R}_{\mathtt{r}_\ell} &= \{R \mid (s, s') \in R \implies s\,\ell = s'\,\ell\} \\
\mathcal{R}_{\mathtt{w}_\ell} &= \{R \mid (s, s') \in R \implies \forall n \in \mathbb{Z}.\, (s[\ell \mapsto n], s'[\ell \mapsto n]) \in R\}
\end{aligned}$$

Apart from the clause for computation types, this is a familiar-looking logical relation. To understand the interpretation of effect annotations, note that the first intersection is intersection of relations, whilst the second is an intersection of *sets* of relations. Then for each ε there is a set $\mathcal{R}_\varepsilon$ of relations on the state that computations of type $T_\varepsilon X$ have to preserve; the more possible effects occur in ε, the fewer relations are preserved.

Thus, for example, if ε is the empty set then $\mathcal{R}_\varepsilon$ is the empty intersection, i.e. *all* state relations. So $[\![T_\emptyset X]\!]$ relates two computations m and m' of type $[\![T(U(X))]\!]$ if for all state relations R and pre-states s,s' related by R, $m\,s$ and $m'\,s'$ yield post-states related by R and values related by $[\![X]\!]$, which is just what one would expect the definition of observational purity to be from the discussion in Section 1.2. A little more calculation shows that, if $\mathcal{L} = \{\mathbf{x}, \mathbf{y}\}$ then a state relation R is in $[\![T_{\{\mathtt{w_x},\mathtt{w_y},\mathtt{r_y}\}}(\mathtt{unit})]\!]$, the interpretation of commands not reading $\mathbf{x}$, just when it (is either empty or) factors as $R_\mathbf{x} \times \Delta$ with $R_\mathbf{x} \supseteq \Delta$, which again matches the observation in the introduction. What is going on is even clearer if one rephrases the RHS of the implication in the definition of $\mathcal{R}_{\mathtt{r}_\ell}$ as

$$([\![\mathtt{read}(\ell)]\!]\,()\,s, [\![\mathtt{read}(\ell)]\!]\,()\,s') \in R \times [\![\mathtt{int}]\!]$$

and that of $\mathcal{R}_{\mathtt{w}_\ell}$ as saying

$$([\![\mathtt{write}(\ell, V)]\!]\,()\,s, [\![\mathtt{write}(\ell, V')]\!]\,()\,s') \in R \times [\![\mathtt{unit}]\!]$$

for all V,V' such that $([\![V]\!]\,(), [\![V']\!]\,()) \in [\![\mathtt{int}]\!]$. The usual 'logical' relational interpretation of a type can be understood as 'preserving all the relations that are preserved by all the operations on the type' and the above shows how the semantics of our refined types really does extend the usual notion: the refined type is a subtype with only a subset of the original operations and thus will preserve all relations that are preserved by that smaller set of operations.

We also extend the relational interpretation of refined types to refined contexts in the natural way:

$$\begin{aligned} [\![\Theta]\!] &\subseteq [\![U(\Theta)]\!] \times [\![U(\Theta)]\!] \\ [\![x_1 : X_1, \ldots, x_n : X_n]\!] &= [\![X_1]\!] \times \cdots \times [\![X_n]\!] \end{aligned}$$

Lemma 5. *For any Θ, X and ε, all of $[\![\Theta]\!]$, $[\![X]\!]$ and $[\![T_\varepsilon X]\!]$ are partial equivalence relations.* □

The following sanity check says that the interpretation of a refined type with the top effect annotation everywhere is just equality on the interpretation of its erasure:

Lemma 6. *For all A, $[\![G(A)]\!] = \Delta_{[\![A]\!]}$.* □

The following establishes semantic soundness for our subtyping relation:

Lemma 7. *If $X \leq Y$ then $[\![X]\!] \subseteq [\![Y]\!]$, and similarly for computation types.* □

And we can then show a 'fundamental theorem' establishing the soundness of the effect analysis itself:

Theorem 1

1. *If $\Theta \vdash V : X$, $(\rho, \rho') \in [\![\Theta]\!]$ then*

$$([\![U(\Theta) \vdash V : U(X)]\!]\,\rho,\ [\![U(\Theta) \vdash V : U(X)]\!]\,\rho') \in [\![X]\!]$$

2. If $\Theta \vdash M : T_\varepsilon X$*,* $(\rho, \rho') \in [\![\Theta]\!]$ *then*

$$([\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho, [\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho') \in [\![T_\varepsilon X]\!] \qquad \square$$

Because we have used standard technology (logical relations, PERs), the pattern of what we have to prove here is obvious and the definitions are all set up so that the proofs go through smoothly. Had we defined the semantics of effects in some more special-purpose way (e.g. trying to work directly with the property of being uniformly either constant or the identity), it could have been rather less clear how to make everything extend smoothly to higher-order and how to deal with combining effects in the let-rule.

2.4 Basic Equations

Before looking at effect-dependent equivalences, we note that the semantics validates all the usual equations of the computational metalanguage, including congruence laws and β and η laws for products, function spaces, booleans and the computation type constructor. We show some of these rules in Figure 4. Note that the correctness of the basic congruence laws subsumes Theorem 1 and that, rather subtly, we have made the reflexivity PER rule invertible. This is sound because our effect annotations are purely descriptive (or *extrinsic* in Reynolds's terminology [17]) whereas the simple types are more conventionally prescriptive (which Reynolds calls *intrinsic*). We actually regard the rules of Figure 3 as abbreviations for a subset of the equational judgements of Figure 4; thus we can allow the refined type of the conclusion of interesting equational rules (e.g. the dead computation rule, to be presented shortly) to be different from (in particular, have a smaller effect than) the refined types in the assumptions. In practical terms, this is important for allowing inferred effects to be improved locally as transformations are performed, rather than having to periodically reanalyse the whole program to obtain the best results.

3 Using Effect Information

More interesting equivalences are predicated on the effect information. The *read-set* of an effect ε is denoted $\mathrm{rds}(\varepsilon)$ and defined as $\{\ell \in \mathcal{L} \mid \mathtt{r}_\ell \in \varepsilon\}$. Likewise, the *write-set* of an effect ε is denoted $\mathrm{wrs}(\varepsilon)$ and defined as $\{\ell \in \mathcal{L} \mid \mathtt{w}_\ell \in \varepsilon\}$. The set of locations mentioned in an effect is $\mathrm{locs}(\varepsilon) = \mathrm{rds}(\varepsilon) \cup \mathrm{wrs}(\varepsilon)$. We make use of these definitions in refined side-conditions for the effect-dependent equivalences, presented in Figure 5.

The **Dead Computation** transformation allows the removal of a computation producing an unused value, provided the effect of that computation is at most reading (if the computation could write the store then its removal would generally be unsound, as that write could be observed by the rest of the computation).

The **Duplicated Computation** transformation allows two evaluations of the same computation to be replaced by one, provided that the observable reads and observable writes of the computation are disjoint. Intuitively, the locations that

PER rules (+ similar for computations):

$$\begin{array}{c} \Theta \vdash V : X \\ \hline\hline \Theta \vdash V = V : X \end{array} \qquad \frac{\Theta \vdash V = V' : X}{\Theta \vdash V' = V : X} \qquad \frac{\Theta \vdash V = V' : X \quad \Theta \vdash V' = V'' : X}{\Theta \vdash V = V'' : X}$$

$$\frac{\Theta \vdash V = V' : X \quad X \leq X'}{\Theta \vdash V = V' : X'}$$

Congruence rules (extract):

$$\frac{\Theta \vdash V_1 = V_1' : \mathtt{int} \quad \Theta \vdash V_2 = V_2' : \mathtt{int}}{\Theta \vdash (V_1 + V_2) = (V_1' + V_2') : \mathtt{int}} \qquad \frac{\Theta \vdash V = V' : X_1 \times X_2}{\Theta \vdash \pi_i\, V \;=\; \pi_i\, V' : X_i}$$

$$\frac{\Theta, x : X \vdash M = M' : T_\varepsilon Y}{\Theta \vdash (\lambda x : U(X).M) = (\lambda x : U(X).M') : X \to T_\varepsilon Y}$$

β rules (extract):

$$\frac{\Theta, x : X \vdash M : T_\varepsilon Y \quad \Theta \vdash V : X}{\Theta \vdash (\lambda x : U(X).M)\, V \;=\; M[V/x] : T_\varepsilon Y} \qquad \frac{\Theta \vdash V : X \quad \Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \mathtt{let}\, x \Leftarrow \mathtt{val}\, V\, \mathtt{in}\, M = M[V/x] : T_\varepsilon Y}$$

η rules (extract):

$$\frac{\Theta \vdash V : X \to T_\varepsilon Y}{\Theta \vdash V = (\lambda x : U(X).V\, x) : X \to T_\varepsilon Y} \qquad \frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash (\mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, \mathtt{val}\, x) = M : T_\varepsilon X}$$

Commuting conversions:

$$\frac{\Theta \vdash M : T_{\varepsilon_1} Y \quad \Theta, y : Y \vdash N : T_{\varepsilon_2} X \quad \Theta, x : X \vdash P : T_{\varepsilon_3} Z}{\Theta \vdash \mathtt{let}\, x \Leftarrow (\mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, N)\, \mathtt{in}\, P = \mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, \mathtt{let}\, x \Leftarrow N\, \mathtt{in}\, P : T_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} Z}$$

Fig. 4. Effect-independent equivalences

may be read on the second evaluation were not written during the first one, so will have the same values. Hence the actual values written during the second evaluation will be the same as were written during the first evaluation. Thus both the final state and the computed value after the second evaluation will be the same (equivalent) to the state and value after the first evaluation.

The **Commuting Computations** transformation allows the order of two value-independent computations to be swapped provided that their write sets are disjoint and neither may read a location that the other may write.

The **Pure Lambda Hoist** transformation allows a computation to be hoisted out of a lambda abstraction (so it is performed once, rather than every time the function is applied) provided that it is observably pure (and, of course, that it does not depend on the function argument). This is not only useful, but also interesting as an example of a transformation we did *not* manage to prove sound in our earlier [6] work.

Dead Computation:

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N = N : T_{\varepsilon'} Y}\; x \notin \Theta, \mathrm{wrs}(\varepsilon) = \emptyset$$

Duplicated Computation:

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X, y : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \begin{array}{l}\texttt{let}\, x \Leftarrow M\, \texttt{in}\, \texttt{let}\, y \Leftarrow M\, \texttt{in}\, N \\ = \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N[x/y]\end{array} : T_{\varepsilon \cup \varepsilon'} Y}\; \mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset$$

Commuting Computations:

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X_1 \quad \Theta \vdash M_2 : T_{\varepsilon_2} X_2 \quad \Theta, x_1 : X_1, x_2 : X_2 \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \begin{array}{l}\texttt{let}\, x_1 \Leftarrow M_1\, \texttt{in}\, \texttt{let}\, x_2 \Leftarrow M_2\, \texttt{in}\, N \\ = \texttt{let}\, x_2 \Leftarrow M_2\, \texttt{in}\, \texttt{let}\, x_1 \Leftarrow M_1\, \texttt{in}\, N\end{array} : T_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon'} Y}\; \begin{array}{l}\mathrm{rds}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset \\ \mathrm{wrs}(\varepsilon_1) \cap \mathrm{rds}(\varepsilon_2) = \emptyset \\ \mathrm{wrs}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset\end{array}$$

Pure Lambda Hoist:

$$\frac{\Theta \vdash M : T_{\{\}} Z \quad \Theta, x : X, y : Z \vdash N : T_\varepsilon Y}{\Theta \vdash \begin{array}{l}\texttt{val}\, (\lambda x : U(X).\texttt{let}\, y \Leftarrow M\, \texttt{in}\, N) \\ = \texttt{let}\, y \Leftarrow M\, \texttt{in}\, \texttt{val}\, (\lambda x : U(X).N)\end{array} : T_{\{\}}(X \to T_\varepsilon Y)}$$

Fig. 5. Effect-dependent equivalences

The following Lemma states that a computation with effect ε cannot change the state of locations outside $\mathrm{wrs}(\varepsilon)$. We write $s =_L s'$ if for all $\ell \in L$, $s(\ell) = s'(\ell)$.

Lemma 8 (No writes). *Suppose $\Theta \vdash M : T_\varepsilon X$ and $(\rho, \rho) \in [\![\Theta]\!]$. If $[\![\Theta \vdash M : T_\varepsilon X]\!]\, \rho\, s_0 = (s_1, x)$ then $s_0 =_{\mathcal{L} \setminus \mathrm{wrs}(\varepsilon)} s_1$.*

Proof. Define a relation $R = \{(s, s) \mid s =_{\mathcal{L} \setminus \mathrm{wrs}(\varepsilon)} s_0\}$. It is easy to see that $R \in \mathcal{R}_\varepsilon$, and clearly $(s_0, s_0) \in R$. Then applying Theorem 1 to M and $(\rho, \rho) \in [\![\Theta]\!]$ we can deduce that $(s_1, s_1) \in R$, so $s_1 =_{\mathcal{L} \setminus \mathrm{wrs}(\varepsilon)} s_0$. □

Dually, running a computation with effect ε on states that differ only outside $\mathrm{rds}(\varepsilon)$ makes an identical change to each state:

Lemma 9 (No reads). *Suppose $\Theta \vdash M : T_\varepsilon X$ and $(\rho, \rho') \in [\![\Theta]\!]$. Let s_0 and s'_0 be two states such that $s_0 =_{\mathrm{rds}(\varepsilon)} s'_0$. If $[\![\Theta \vdash M : T_\varepsilon X]\!]\, \rho\, s_0 = (s_1, x)$ and $[\![\Theta \vdash M : T_\varepsilon X]\!]\, \rho'\, s'_0 = (s'_1, x')$ then $(x, x') \in [\![X]\!]$ and for all $\ell \in \mathcal{L}$, either $s_1(\ell) = s'_1(\ell)$ (locations are updated, identically), or $s_1(\ell) = s_0(\ell)$ and $s'_1(\ell) = s'_0(\ell)$ (locations are left unchanged).*

Proof. Define a relation $R = \{(s, s') \mid \forall \ell \in \mathcal{L}, s\,\ell = s'\,\ell \vee (s\,\ell = s_0\,\ell \wedge s'\,\ell = s'_0\,\ell)\}$. It is straightforward to check that $R \in \mathcal{R}_\varepsilon$, and that $(s_0, s'_0) \in R$. Then applying Theorem 1 to M and $(\rho, \rho') \in [\![\Theta]\!]$ we can deduce that $(s_1, s'_1) \in R$ and $(x, x') \in [\![X]\!]$ and the result follows immediately. □

If $U(\Theta) \vdash V : U(X)$ and $U(\Theta) \vdash V' : U(X)$ then write $\Theta \models V = V' : X$ to mean that for all $(\rho, \rho') \in [\![\Theta]\!]$

$$([\![U(\Theta) \vdash V : U(X)]\!]\, \rho,\ [\![U(\Theta) \vdash V' : U(X)]\!]\, \rho') \in [\![X]\!]$$

and similarly for computations.

Theorem 2. *All of the equations shown in Figures 4 and 5 are soundly modelled in the semantics:*

- *If* $\Theta \vdash V = V' : X$ *then* $\Theta \models V = V' : X$.
- *If* $\Theta \vdash M = M' : T_\varepsilon X$ *then* $\Theta \models M = M' : T_\varepsilon X$.

Proof. We present proofs for the equivalences in Figure 5.

Dead computation. If we let $\Gamma = U(\Theta)$, $A = U(X)$ and $B = U(Y)$ and $(\rho, \rho') \in [\![\Theta]\!]$ then we have to show

$$([\![\Gamma \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N : TB]\!]\, \rho,\ [\![\Gamma \vdash N : TB]\!]\, \rho') \in [\![T_{\varepsilon'} Y]\!]$$

Pick $R \in \mathcal{R}_{\varepsilon'}$ and $(s, s') \in R$, and let $(s_1, x) = [\![\Gamma \vdash M : TA]\!]\, \rho\, s$. As $[\![\Theta]\!]$ is a PER we know $(\rho, \rho) \in [\![\Theta]\!]$, and because $\mathrm{wrs}(\varepsilon) = \emptyset$ we can apply Lemma 8 to deduce that $s_1 = s$. Hence

$$[\![\Gamma \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N : TB]\!]\, \rho\, s = [\![\Gamma \vdash N : TB]\!]\, \rho\, s$$

and by assumption on N

$$([\![\Gamma \vdash N : TB]\!]\, \rho\, s, [\![\Gamma \vdash N : TB]\!]\, \rho'\, s') \in R \times [\![Y]\!]$$

so we're done.

Pure lambda hoist. Define $\Gamma = U(\Theta)$, $A = U(X)$, $B = U(Y)$, $C = U(Z)$. Pick $(\rho, \rho') \in [\![\Theta]\!]$, $R \in \mathcal{R}_{\{\}}$ and $(s, s') \in R$ (note that R is actually unconstrained in this case). Then

$$\begin{aligned} &[\![\Gamma \vdash \texttt{val}\, (\lambda x : A.\texttt{let}\, y \Leftarrow M\, \texttt{in}\, N) : T(A \to TB)]\!]\, \rho\, s \\ &= (s, \lambda x \in [\![A]\!].[\![\Gamma, x : A \vdash \texttt{let}\, y \Leftarrow M\, \texttt{in}\, N : TB]\!]\, (\rho, x)) \end{aligned}$$

and

$$\begin{aligned} &[\![\Gamma \vdash \texttt{let}\, y \Leftarrow M\, \texttt{in}\, \texttt{val}\, (\lambda x : A.N) : T(A \to TB)]\!]\, \rho', s' \\ &= (s'', \lambda x' \in [\![A]\!].[\![\Gamma, x : A, y : C \vdash N : TB]\!]\, (\rho', x', y')) \end{aligned}$$

where

$$(s'', y') = [\![\Gamma \vdash M : TC]\!]\, \rho'\, s'$$

Now, as M doesn't write, Lemma 8 entails $s'' = s'$ and hence $(s, s'') \in R$. Thus it remains to show that the two functions are in $[\![X \to T_\varepsilon Y]\!]$. So assume $(x, x') \in [\![X]\!]$, we have now to show

$$\begin{matrix} ([\![\Gamma, x : A \vdash \texttt{let}\, y \Leftarrow M\, \texttt{in}\, N : TB]\!]\, (\rho, x), \\ [\![\Gamma, x : A, y : C \vdash N : TB]\!]\, (\rho', x', y')) \end{matrix} \in [\![T_\varepsilon Y]\!]$$

So pick $R_2 \in \mathcal{R}_\varepsilon$, $(s_2, s_2') \in R_2$ and calculate

$$[\![\Gamma, x{:}A \vdash \texttt{let}\, y \Leftarrow M \,\texttt{in}\, N : TB]\!]\,(\rho, x)\, s_2 = [\![\Gamma, x{:}A, y{:}C \vdash N : TB]\!]\,(\rho, x, y_2)\, s_3$$

where (as $x \notin fv(M)$)

$$(s_3, y_2) = [\![\Gamma \vdash M : TC]\!]\,\rho\, s_2$$

By Lemma 8, $s_3 = s_2$, so $(s_3, s_2') \in R_2$. As M preserves all relations, it preserves $\{(s_2, s')\}$, so $(y_2, y') \in [\![Z]\!]$, which implies

$$((\rho, x, y_2), (\rho', x', y')) \in [\![\Theta, x : X, y : Z]\!]$$

so we're done by assumption that N preserves R_2.

Duplicated computation. Let $\Gamma = U(\Theta)$, $A = U(X)$, $B = U(Y)$ and $(\rho, \rho') \in [\![\Theta]\!]$. Because $[\![\Theta]\!]$ is a PER, we also have $(\rho, \rho) \in [\![\Theta]\!]$ and $(\rho', \rho') \in [\![\Theta]\!]$. Pick $R \in \mathcal{R}_{\varepsilon \cup \varepsilon'}$ and $(s_0, s_0') \in R$. We need to show

$$\begin{array}{l}([\![\Gamma \vdash \texttt{let}\, x \Leftarrow M; y \Leftarrow M \,\texttt{in}\, N : TB]\!]\,\rho\, s_0, \\ \;[\![\Gamma \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N[x/y] : TB]\!]\,\rho'\, s_0')\end{array} \in R \times [\![Y]\!]$$

Let

$$\begin{aligned}(s_1, x) &= [\![\Gamma \vdash M : TA]\!]\,\rho\, s_0 \\ (s_1', x') &= [\![\Gamma \vdash M : TA]\!]\,\rho'\, s_0' \\ (s_2, y) &= [\![\Gamma \vdash M : TA]\!]\,\rho\, s_1.\end{aligned}$$

By Lemma 8 we can deduce $s_1 =_{\mathcal{L} \setminus \mathrm{wrs}(\varepsilon)} s_0$. We can use this fact as assumption to Lemma 9 starting in states s_1 and s_0, since $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset$, to obtain $(y, x) \in [\![X]\!]$ and for all $\ell \in \mathcal{L}$, either $s_2(\ell) = s_1(\ell)$, or $s_2(\ell) = s_1(\ell)$ and $s_1(\ell) = s_0(\ell)$. Hence $s_2 = s_1$; in other words, M behaves idempotently.

Expanding the semantics,

$$\begin{aligned}[\![\Gamma \vdash \texttt{let}\, x \Leftarrow M; y \Leftarrow M \,\texttt{in}\, N : TB]\!]\,\rho\, s_0 &= [\![\Gamma' \vdash N : TB]\!]\,(\rho, x, y)\, s_2 \\ [\![\Gamma \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N[x/y] : TB]\!]\,\rho'\, s_0' &= [\![\Gamma' \vdash N : TB]\!]\,(\rho', x', x')\, s_1'\end{aligned}$$

where $\Gamma' = \Gamma, x : A, y : A$.

Since $R \in \mathcal{R}_{\varepsilon \cup \varepsilon'}$ we must have $R \in \mathcal{R}_\varepsilon$. Therefore M preserves R, so we can deduce that $(x, x') \in [\![X]\!]$ and $(s_1, s_1') \in R$, so $(s_2, s_1') \in R$. By transitivity we have that $(y, x') \in [\![X]\!]$. Hence $((\rho, x, y), (\rho', x', x')) \in [\![\Gamma']\!]$. Finally, because $R \in \mathcal{R}_{\varepsilon'}$, we know that N preserves R, from which we obtain the desired result.

Commuting computations. Let $\Gamma = U(\Theta)$, $A_i = U(X_i)$ and $B = U(Y)$. Pick $(\rho, \rho') \in [\![\Theta]\!]$, $R \in \mathcal{R}_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon'}$, $(s_0, s_0') \in R$. Let

$$\begin{array}{rcl}(s_1, x_1) = [\![\Gamma \vdash M_1 : TA_1]\!]\,\rho\, s_0 & \text{and} & (s_2, x_2) = [\![\Gamma \vdash M_2 : TA_2]\!]\,\rho\, s_1 \\ (s_1', x_2') = [\![\Gamma \vdash M_2 : TA_2]\!]\,\rho'\, s_0' & \text{and} & (s_2', x_1') = [\![\Gamma \vdash M_1 : TA_1]\!]\,\rho'\, s_1'.\end{array}$$

By the definition of $\mathcal{R}_{\varepsilon_1 \cup \varepsilon_2}$ for reading we know (1) that $s_0 =_{\mathrm{rds}(\varepsilon_1)\cup\mathrm{rds}(\varepsilon_2)} s'_0$. By four applications of Lemma 8, we have

$$\begin{array}{llll} s_1 =_{\mathcal{L}\setminus\mathrm{wrs}(\varepsilon_1)} s_0 & (2) & s_2 =_{\mathcal{L}\setminus\mathrm{wrs}(\varepsilon_2)} s_1 & (3) \\ s'_1 =_{\mathcal{L}\setminus\mathrm{wrs}(\varepsilon_2)} s'_0 & (4) & s'_2 =_{\mathcal{L}\setminus\mathrm{wrs}(\varepsilon_1)} s'_1 & (5) \end{array}$$

From (1), (4) and the first side-condition on the rule, we have $s'_0 =_{\mathrm{rds}(\varepsilon_1)} s'_1$. We can use this as assumption to apply Lemma 9 to M_1 starting in states s_0 and s'_1 with corresponding environments ρ and ρ', to get $(x_1, x'_1) \in [\![X_1]\!]$ and

$$\forall \ell \in \mathcal{L}, s_1(\ell) = s'_2(\ell) \vee (s_1(\ell) = s_0(\ell) \wedge s'_2(\ell) = s'_1(\ell)) \tag{6}$$

From (1), (2) and the second side-condition on the rule, we have $s'_0 =_{\mathrm{rds}(\varepsilon_2)} s_1$. We can use this as assumption to apply Lemma 9 to M_2 starting in states s_1 and s'_0 with corresponding environments ρ and ρ', to get $(x_2, x'_2) \in [\![X_2]\!]$ and

$$\forall \ell \in \mathcal{L}, s_2(\ell) = s'_1(\ell) \vee (s_2(\ell) = s_1(\ell) \wedge s'_1(\ell) = s'_0(\ell)) \tag{7}$$

We now show that for all $\ell \in \mathcal{L}$ either $\ell \in \mathrm{wrs}(\varepsilon_1 \cup \varepsilon_2)$ and $s_2(\ell) = s'_2(\ell)$, or $s_2(\ell) = s_0(\ell)$ and $s'_2(\ell) = s'_0(\ell)$. In other words, there is some state change Δ with $\mathrm{dom}(\Delta) \subseteq \mathrm{wrs}(\varepsilon_1 \cup \varepsilon_2)$ such that $s_2 = s_0[\Delta]$ and $s'_2 = s'_0[\Delta]$.

First suppose $\ell \notin \mathrm{wrs}(\varepsilon_1 \cup \varepsilon_2)$. By (2) and (3) we have $s_2(\ell) = s_0(\ell)$, and by (4) and (5) we have $s'_2(\ell) = s'_0(\ell)$ so we've shown the right hand disjunct.

Now suppose $\ell \in \mathrm{wrs}(\varepsilon_1)$. Therefore $\ell \notin \mathrm{wrs}(\varepsilon_2)$ by the third side-condition on the rule. By (6) either $s_1(\ell) = s'_2(\ell)$ $(= s_2(\ell)$ by (3)), or $s_1(\ell) = s_0(\ell)$ $(= s_2(\ell)$ by (3)) and $s'_2(\ell) = s'_1(\ell)$ $(= s'_0(\ell)$ by (4)) which is the disjunction above. Similar reasoning applies if $\ell \in \mathrm{wrs}(\varepsilon_2)$.

Since $(s_0, s'_0) \in R$ we can show that $(s_2, s'_2) \in R$ by induction on the size of $\mathrm{dom}(\Delta)$, using the definition of $\mathcal{R}_{\mathtt{w}_\ell}$ for each $\ell \in \mathrm{dom}(\Delta)$.

Now, expanding the semantics,

$$\begin{array}{r} [\![\Gamma \vdash \mathtt{let}\, x_1 \Leftarrow M_1; x_2 \Leftarrow M_2 \;\mathtt{in}\; N : TB]\!]\, \rho\, s_0 = [\![\Gamma' \vdash N : TB]\!]\, (\rho, x_1, x_2)\, s_2 \\ [\![\Gamma \vdash \mathtt{let}\, x_2 \Leftarrow M_2; x_1 \Leftarrow M_1 \;\mathtt{in}\; N : TB]\!]\, \rho'\, s'_0 = [\![\Gamma' \vdash N : TB]\!]\, (\rho', x'_1, x'_2)\, s'_2 \end{array}$$

where $\Gamma' = \Gamma, x_1 : A_1, x_2 : A_2$. We have $((\rho, x_1, x_2), (\rho', x'_1, x'_2)) \in [\![\Gamma']\!]$. Finally, because $R \in \mathcal{R}_{\varepsilon'}$, we know that N preserves R starting in states s_2 and s'_2, from which we obtain the desired result. □

To make the link between relatedness and contextual equivalence, we have to say something just a little more sophisticated than 'related terms are contextually equivalent', as we also have to restrict the set of contexts. Write $(\Theta \vdash T_\varepsilon X)^\top$ for the set of all ground contexts $C[-]$ whose holes $-$ are typable as $\Theta \vdash - : T_\varepsilon X$ in the extended language. Then write $(\Theta \models T_\varepsilon X)^\top$ for the set of all contexts with a hole typeable as $U\Theta \vdash - : T(UX)$ in the base language such that

$$\forall M, M'.\, \Theta \models M = M' : T_\varepsilon X \implies [\![\vdash C[M] : T(\mathtt{bool})]\!] = [\![\vdash C[M'] : T(\mathtt{bool})]\!]$$

Then Theorem 2 plus adequacy implies that whenever $\Theta \vdash M = M' : T_\varepsilon X$, then for all $C[-]$ in $(\Theta \models T_\varepsilon X)^\top$ and for all s_0,s_1

$$\langle s_0, C[M]\rangle \Downarrow \langle s_1, \mathit{true}\rangle \iff \langle s_0, C[M']\rangle \Downarrow \langle s_1, \mathit{true}\rangle.$$

and by the congruence rules, $(\Theta \vdash T_\varepsilon X)^\top \subseteq (\Theta \models T_\varepsilon X)^\top$.

The equations above also imply some effect-dependent type isomorphisms, proved by defining contexts transforming typed terms in both directions and showing that both compositions rewrite to the identity. For example

$$X \times Y \to T_\varepsilon Z \;\cong\; X \to T_{\{\}}(Y \to T_\varepsilon Z)$$

follows from $\beta\eta$ rules and the pure lambda hoist equation. However, there are valid contextual equivalences and isomorphisms that do not follow from the semantics. For example

$$(1 \to T_{\mathtt{W_x}}\mathtt{bool}) \to T_{\{\}}\mathtt{bool} \;\cong\; 1 \to T_{\{\}}\mathtt{bool}$$

does not hold in the model because of the presence of the non-definable 'snap-back' [7] function $\lambda g.\lambda s.\, let\ (s', b) = g()\, s\ in\ (s, b)$.

4 Discussion

We have shown how an extensional interpretation of read and write effects may be given using a non-standard form of relational parametricity over a standard semantics, and how that semantics may be used to justify program transformations. This contrasts with more common intensional approaches, based on traces in an instrumented semantics, which fail to decouple program properties from a particular syntactic system for establishing them and are not well-suited to reasoning about equivalences. We have also verified the interesting results of Sections 2.3 and 3 using the Coq proof assistant; the script is available via the first author's homepage.

The general relational approach that we are using here has been demonstrated to work well in both denotational and operational settings. Denotational approaches to the semantics of analysis properties using directly the obvious "factors through" style of definition (e.g. saying a computation is observationally pure if its denotation factors through that of `val`) can easily raise unpleasant questions of definability if one tries to recast them in an operational framework.

In this paper we have concentrated on an extremely simple effect system, so as to make the methodology as clear as possible. Working with domains instead of sets, to allow recursion, is straightforward. With Buchlovsky, we have also successfully applied just the same techniques to reason about transformations justified by an effect analysis for exceptions. Looking at the set of all relations preserved by a subset of the operations on a monad really does seem to be the 'right' way of understanding effect systems (and seems not unrelated to the algebraic view of effects being developed by Plotkin and Power [15]). We are confident the idea extends to a wide class of effect analyses (and more general notions of refinement type), and are currently working on applying it to region-based encapsulation of state effects in the presence of dynamic allocation. This is a challenging problem; despite much work on monadic encapsulation (and on region-based memory management [19]) since the introduction of `runST` in Haskell [10], some of it incorrect and most of it rather complex, previous work mostly addresses simple syntactic type soundness, rather than equations [12], though the region calculus has been given a relation-based semantics [2] and studied using bisimulation [9]. Parametric logical relations have previously been used for establishing particular equivalences involving encapsulated state [16,7] and even provide a complete characterization of contextual equivalence for a language with integer store [14]. However, a combination of those constructions with our notion of refined types that is suitably generic and also expressive enough to validate, for example, interesting cases of the duplicated computations equation, has so far proved elusive.[2]

One interesting application of effect analyses is in assertion checking for imperative languages. Assertions are typically boolean expressions in the same language as is being checked and make use of side-effecting operations such as mutation in computing their results. Yet it is important that these side-effects do not affect the behaviour of the program being specified: assertions should be observationally pure. Naumann uses simulation relations to capture a notion of observational purity for boolean-valued expressions that allows mutation of encapsulated state [13].

PER-based accounts of dependency and information flow [1,18] are closely related to the present work; as a referee observed, observable notions of reading and writing have a natural connection with confidentiality and integrity.

Apart from the lines of future work implicit in the above, it would be interesting to try to use our approach to capture some general relationship between effect systems and their intuitive duals, capability/permission systems.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *26th Symposium on Principles of Programming Languages (POPL)*, 1999.
2. 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.
3. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, January 2004. Revised version available from `http://research.microsoft.com/~nick/publications.htm`.
4. N. Benton. Semantics of program analyses and transformations. Lecture Notes for the PAT Summer School, Copenhagen, June 2005.
5. N. Benton, J. Hughes, and E. Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
6. N. Benton and A. Kennedy. Monads, effects and transformations. In *3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), Paris*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999.

[2] This is not *quite* the same problem as in region-based memory management. We want to reason about encapsulated state being non-observable to the rest of the program, but that does not necessarily mean it may safely be deallocated.

7. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.
8. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, Massachusetts, August 1986.
9. S. Helsen. Bisimilarity for the region calculus. *Higher-Order and Symbolic Computation*, 17(4), 2004.
10. S. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4), 1995.
11. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1988.
12. E. Moggi and A. Sabry. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming*, 11(6), 2001.
13. D. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, To appear.
14. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.
15. G. D. Plotkin and J. Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures, Proceedings of FOSSACS '02*, volume 2303 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
16. 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.
17. J. C. Reynolds. The meaning of types – from intrinsic to extrinsic semantics. Technical Report BRICS RS-00-32, BRICS, University of Aarhus, December 2000.
18. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.
19. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. A. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Proceedings of the Workshop on Types in Compilation (TIC)*, volume 1473 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
21. P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.