

# Correctness of effect-based program transformations

Martin Hofmann

*Institut für Informatik, LMU München*

**Abstract.** We consider a type system capable of tracking reading, writing and allocation in a higher-order language with dynamically allocated references.

We give a denotational semantics to this type system which allows us to validate a number of effect-dependent program equivalences in the sense of observational equivalence. An example is the following:

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

provided that  $e$  does not read from memory regions that it writes to and moreover does not allocate memory that is encapsulated in the values of  $x$  and  $y$ .

Here  $x$  can be a higher-order function or a reference or a combination of both.

The two sides of the above equivalence turn out to be related in the denotational semantics which implies that they are observationally equivalent, ie can be replaced by one another in any (well-typed) program.

On the way we learn popular techniques such as parametrised logical relations, regions, admissible relations, etc., which belong to the toolbox of researchers in principles of programming languages.

**Keywords.** program transformation, denotational semantics, correctness of programs, logical relations.

## 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

$$\begin{aligned} C ; \text{if } B \text{ then } C' \text{ else } C'' &= \\ \text{if } B \text{ then } (C;C') \text{ else } (C;C'') \end{aligned}$$

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 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 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.

We begin by defining a simple language with global integer references and describe an effect typing system allowing us to track reading to and writing from individual locations. We then state a list of effect-dependent program equivalences whose correctness with respect to observational equivalence we then embark on proving. To do this, we develop a relational semantics which models effects as sets of relations that are preserved by computations exhibiting that effect. The more side effects the fewer relations are preserved. In particular, if an operation may read location  $\ell$  then only those relations  $R$  for which  $sRs'$  implies  $s(\ell) = s'(\ell)$  can be preserved. If an operation writes  $\ell$  then only those relations  $R$  for which  $sRs'$  implies  $s[\ell:=n]Rs'[\ell:=n]$  for all  $n$  can be preserved. The relational semantics then defines a partial equivalence relation between values of the same given type which is shown to imply observational equivalence and at the same time to include the equational theory generated by our list of effect-dependent program transformations which therefore are valid with respect to observational equivalence.

We then extend this basic framework to encompass dynamically allocated references, recursive definitions, references of structured and even functional types (the latter two not contained in these lecture notes, though). Each of these extensions requires new methods such as domains, partial bijections, Kripke logical relations, which are of independent interest. With dynamically allocated references manifest effects can sometimes be discounted from the analysis on the grounds that they affect only “private” portions of the store, a phenomenon known as *effect masking*. We thus also explain effect masking semantically and show how it helps us to justify more program transformations.

These notes are based on joint work with Lennart Beringer, Nick Benton, and Andrew Kennedy; a large portion of the material is from our joint publications [5,4]; Quasi-PERS in the context of logical relations appear here for the first time.

## 2. Syntax

Types are given by the following grammar:

$A, B ::= \text{unit} \mid \text{int} \mid \text{bool} \mid \text{ref} \mid A \times B \mid A \rightarrow B$

We assume an infinite supply  $\mathbb{L}$  of locations ranged over by  $\ell$ , possibly decorated, and an infinite supply of variables ranged over by  $x, y, z$  possibly decorated. In concrete examples we may also use other identifiers for variables.

Terms ( $e$ ) are given by the grammar:

$$e ::= x \mid n \mid \ell \mid \text{true} \mid \text{false} \mid e_1 \text{ op } e_2 \mid () \mid (e_1, e_2) \mid e.1 \mid e.2 \mid e_1 \ e_2 \mid \\ \lambda x. e \mid \text{let } x \leftarrow e_1 \text{ in } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid !e \mid e_1 := e_2$$

Here  $n$  ranges over integer constants and  $op$  ranges over a suitable set of binary operators including arithmetic operations and comparisons.

The expression  $!e$  denotes the value contained in the reference (denoted by)  $e$  and  $e_1 := e_2$  denotes assignment. The type **unit** has exactly one element denoted  $()$ . The type  $A \times B$  is the cartesian product of  $A$  and  $B$ , its elements are pairs  $(x, y)$  where  $x : A$  and  $y : B$ . The components of such a pair are accessed with the projections  $.1$  and  $.2$ . In examples, we also use products with more than one two factors whose components are then accessed with  $.1, .2, .3$ , etc. We use the abbreviation

$$e_1; e_2 \stackrel{\text{def}}{=} \text{let } x \leftarrow e_1 \text{ in } e_2$$

when  $x$  is not free in  $e_2$ .

### 2.1. Example programs

The following example programs illustrate the language concepts; we give them here with their types informally anticipating the typing rules from the next subsection.

$$\text{ASSIGNER} \stackrel{\text{def}}{=} \lambda x. \ell := x : \text{int} \rightarrow \text{unit}$$

This assigns a given value to the fixed location  $\ell$ .

$$\text{COUNTER} \stackrel{\text{def}}{=} \lambda x. (\lambda u. x := !x + 1, \lambda u. !x, \lambda u. x := 0) : \\ \text{ref} \rightarrow (\text{unit} \rightarrow \text{unit}) \times (\text{unit} \rightarrow \text{int}) \times (\text{unit} \rightarrow \text{unit})$$

The function *COUNTER* takes a reference as an argument and returns a “counter object” comprising methods for incrementing, getting the current value of, and resetting that reference.

$$\text{MEMO} \stackrel{\text{def}}{=} \lambda l_1. \lambda l_2. \lambda f. \\ l_1 := 0; l_2 := f(0); \\ \lambda x. \text{if } x = !l_1 \text{ then } !l_2 \text{ else} \\ \text{let } u \leftarrow f \ x \text{ in } l_1 := x; l_2 := u; u : \\ \text{ref} \rightarrow \text{ref} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$$

$$\begin{array}{c}
\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \overline{\Gamma \vdash \ell : \text{ref}} \\
\frac{\overline{\Gamma \vdash n : \text{int}} \quad \overline{\Gamma \vdash () : \text{unit}} \quad \overline{\Gamma \vdash \text{true} : \text{bool}} \quad \overline{\Gamma \vdash \text{false} : \text{bool}}}{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B \quad \Gamma \vdash e : A \times B} \\
\frac{\overline{\Gamma \vdash e_1 \text{ op } e_2 : \text{int}} \quad \overline{\Gamma \vdash (e_1, e_2) : A \times B} \quad \overline{\Gamma \vdash e.1 : A}}{\Gamma \vdash e : A \times B \quad \Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A \quad \Gamma, x:A \vdash e : B} \\
\frac{\overline{\Gamma \vdash e.2 : B} \quad \overline{\Gamma \vdash e_1 e_2 : B} \quad \overline{\Gamma \vdash \lambda x.e : A \rightarrow B}}{\Gamma \vdash e_1 : A \quad \Gamma, x:A \vdash e_2 : B \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A \quad \Gamma \vdash e : \text{ref}} \\
\frac{\overline{\Gamma \vdash \text{let } x \leftarrow e_1 \text{ in } e_2 : B} \quad \overline{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : A} \quad \overline{\Gamma \vdash !e : \text{int}}}{\Gamma \vdash e_1 : \text{ref} \quad \Gamma \vdash e_2 : \text{int}} \\
\Gamma \vdash e_1 := e_2 : \text{unit}
\end{array}$$

**Figure 1.** Typing rules without effects

A memo functional. It takes two references and a function  $f$  as arguments. It returns a function  $f'$  which does the same as  $f$  but is arguably more efficient:  $f'$  saves the last argument it has been called with in reference  $l_1$  and puts the corresponding  $f$ -value in  $l_2$ . Thus, if  $f'$  is called several times with the same argument in a row then only the first time a possibly expensive call to  $f$  is launched.

## 2.2. Typing rules

A typing context  $\Gamma$  binds variables to types, we may write it in the form

$$\Gamma := x_1 : A_1, \dots, x_n : A_n$$

Then  $\text{dom}(\Gamma) \stackrel{\text{def}}{=} \{x_1, \dots, x_n\}$  and  $\Gamma(x_i) = A_i$ .

A typing judgement is an assertion of the form  $\Gamma \vdash e : A$  where  $\Gamma$  binds the free variables of  $e$ . It is inductively defined by the typing rules in Figure 1.

## 3. Semantics

We do not give evaluation rules for references but instead show how, even in the presence of references, terms and functions can be understood as mathematical functions. This kind of giving semantics is known as denotational semantics. Note that, for the sake of simplicity, our language does not contain recursion, hence all programs terminate. This simplification allows us to use ordinary total functions on sets rather than continuous functions on Scott domains or similar. It is perfectly possible to add terminating loops such as for-loops, which again we refrain from doing, this time only to keep the syntax small.

We model states as functions from locations to integer values and assign to each type  $A$  a set  $\llbracket A \rrbracket$  by the following clauses:

$$\begin{aligned}
\llbracket \text{int} \rrbracket &= \mathbb{Z} \\
\llbracket \text{unit} \rrbracket &= \{()\} \\
\llbracket \text{bool} \rrbracket &= \{\text{true}, \text{false}\} \\
\llbracket A \times B \rrbracket &= \llbracket A \rrbracket \times \llbracket B \rrbracket \\
\llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \Rightarrow T(\llbracket B \rrbracket) \\
T(X) &= \mathbb{S} \Rightarrow \mathbb{S} \times X \\
\mathbb{S} &= \mathbb{L} \Rightarrow \mathbb{Z}
\end{aligned}$$

The last three clauses deserve some explanation: For sets  $X, Y$  the set  $X \Rightarrow Y$  comprises all functions from  $X$  to  $Y$ . If there were no references we could simply put  $\llbracket A \rightarrow B \rrbracket = \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket$ . But a term such as  $\lambda x. \ell := x$  does not merely return a value but also has a side effect, namely to assign the argument to the reference  $\ell$ . Semantically, this is modelled by the operator  $T(-)$ , a so-called *monad*. Here  $T(X)$  is simply an abbreviation for the set of functions that explain how to get from a given state the new state and the value. Accordingly, elements of  $T(\llbracket A \rrbracket)$  will also serve as denotations of (as yet to be evaluated) terms of type  $A$ . Variables of type  $A$ , on the other hand, will always have values in  $\llbracket A \rrbracket$  because we assume a call-by-value semantics whereby an expression must be fully evaluated before its value can be bound to a variable. Of course, when binding a function to a variable, side effects may be encapsulated in that function as “latent effects” which only come to bear when the function is evaluated. Accordingly, an element of, say  $\llbracket \text{unit} \rightarrow \text{unit} \rrbracket$  does in general refer to the state.

Let  $\Gamma$  be a type context. An environment for  $\Gamma$  is a function  $\eta$  that maps each variable  $x \in \text{dom}(\Gamma)$  to an element  $\eta(x) \in \llbracket \Gamma(x) \rrbracket$ . If now  $\Gamma \vdash e : A$  then we define an element

$$\llbracket e \rrbracket \eta \in T(\llbracket A \rrbracket)$$

by the clauses in Figure 2. Note that  $\llbracket e \rrbracket \eta$  is not an element of  $\llbracket A \rrbracket$  itself because the evaluation of  $e$  may cause side effects and its value may depend on the state. The missing clauses are left as exercises.

*A-normal form* By introducing additional let-expressions it is possible to transform any term into a term in which the term formers  $(-, -)$ ,  $.1$ ,  $.2$ , application,  $!(-)$ ,  $- := -$ , are applied to variables only and such that moreover the guard of a case distinction is a variable. For example, the term  $g(!x)(!f y)$  becomes

```

let u ← !x in
let l ← f y in
let v ← !l in let h ← g u in h v

```

This form has been called administrative normal form (ANF, A normal form) in [7]. If we assume that terms are in ANF the semantic equations and also typing rules can be considerably simplified. For example, we have

$$\begin{aligned}
\llbracket x \rrbracket \eta s &= (s, \eta(x)) \\
\llbracket c \rrbracket \eta s &= (s, c) \\
\llbracket (e_1, e_2) \rrbracket \eta s &= (s_2, (v_1, v_2)) \text{ where } (s_1, v_1) = \llbracket e_1 \rrbracket \eta s, (s_2, v_2) = \llbracket e_2 \rrbracket \eta s_1 \\
\llbracket e_1 e_2 \rrbracket \eta s &= v_1 v_2 s_2 \text{ where } (s_1, v_1) = \llbracket e_1 \rrbracket \eta s, (s_2, v_2) = \llbracket e_2 \rrbracket \eta s_1 \\
\llbracket \lambda x. e \rrbracket \eta s &= (s, f) \text{ where } f(v) = \llbracket e \rrbracket \eta [x \mapsto v] \\
\llbracket \text{let } x \leftarrow e_1 \text{ in } e_2 \rrbracket \eta s &= \llbracket e_2 \rrbracket \eta [x \mapsto v_1] s_1 \text{ where } (s_1, v_1) = \llbracket e_1 \rrbracket \eta s \\
\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket \eta s &= \llbracket e_2 \rrbracket \eta s_1 \text{ when } \llbracket e_1 \rrbracket \eta s = (s_1, \text{true}) \\
\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket \eta s &= \llbracket e_3 \rrbracket \eta s_1 \text{ when } \llbracket e_1 \rrbracket \eta s = (s_1, \text{false}) \\
\llbracket !e \rrbracket &= s_1(\ell) \text{ where } (s_1, \ell) = \llbracket e \rrbracket \eta s \\
\llbracket e_1 := e_2 \rrbracket \eta s &= (s_2[\ell \mapsto v_2], ()) \text{ where } (s_1, \ell) = \llbracket e_1 \rrbracket \eta s, (s_2, v_2) = \llbracket e_2 \rrbracket \eta s_1
\end{aligned}$$

**Figure 2.** Selected semantic equations defining runtime behaviour

$$\begin{aligned}
\llbracket (x, y) \rrbracket \eta s &= (s, (\eta(x), \eta(y))) \\
\llbracket x := y \rrbracket \eta s &= (s[\eta(x) \mapsto \eta(y)], ())
\end{aligned}$$

and so forth. We henceforth assume all terms to be in ANF except in concrete examples.

The ANF also appears in the compilation of functional languages and is closely related to static single assignment (SSA) form known from intermediate language used by compilers.

#### 4. Effect-independent equivalences

If two terms have equal semantics they can be replaced by one another. The equations in Figure 4 hold in this sense where  $v, v_1, v_2$  are *values*, i.e., terms obtained by the following grammar:

$$v ::= x \mid \ell \mid c \mid \lambda x. e \mid (v_1, v_2) \mid v.1 \mid v.2 \mid \text{if } v \text{ then } v_1 \text{ else } v_2$$

If  $v$  is a value then  $\llbracket v \rrbracket \eta s = (s, w)$  for some semantic value  $w$  depending only on  $v$  and  $\eta$  but not on  $s$ . There are several more such generally valid equations involving “let” and “if”. We remark that it is possible to give a complete set of equations that characterise semantic equality assuming that expressions may have arbitrary side effects [3]. Here, however, we are interested in equivalences that might not hold in general, but do hold under extra assumptions on the kinds of side effects that could possibly happen.

$$\begin{aligned}
(\lambda x.e)v &\stackrel{sem}{=} e[x := v] \\
(v_1, v_2).1 &\stackrel{sem}{=} v_1 \\
(v_1, v_2).2 &\stackrel{sem}{=} v_2 \\
v &\stackrel{sem}{=} () \text{ if } v : \mathbf{unit} \\
v &\stackrel{sem}{=} (v.1, v.2) \text{ if } v : A \times B \\
v &\stackrel{sem}{=} \lambda x.v \ x \text{ if } v : A \rightarrow B \\
\mathbf{if } v \mathbf{ then (if } v \mathbf{ then } e_1 \mathbf{ else } e_2) \mathbf{ else } e_2 &\stackrel{sem}{=} \mathbf{if } v \mathbf{ then } e_1 \mathbf{ else } e_2 \\
(\mathbf{let } x \leftarrow e_1 \mathbf{ in } e_2).1 &\stackrel{sem}{=} \mathbf{let } x \leftarrow e_1 \mathbf{ in } e_2.1
\end{aligned}$$

Figure 3. Semantically valid equations

## 5. Monads and metalanguage

The expressions on the left-hand side of the semantic equations (Fig. 2) look very much like terms of a programming language themselves even though they are meant to be “plain English”. One can formalise this and accordingly introduce a *metalanguage* [17] which then allows one to interpret various concrete languages with different features. Indeed, this was one of the initial motivations for the design of the programming language ML (Meta Language) [10].

Such a metalanguage is then a simply-typed lambda calculus with an additional type former  $T$  which can be instantiated according to the kinds of side effects offered by the concrete language to be interpreted. This type former  $T$  must come equipped with constructs **val** and **let** governed by the following typing rules.

$$\frac{\Gamma \vdash e_1 : T(A_1) \quad \Gamma, x:A_1 \vdash e_2 : T(A_2)}{\Gamma \vdash \mathbf{let } x \leftarrow e_1 \mathbf{ in } e_2 : A_2} \quad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathbf{val } e : T(A)}$$

Thus the definition of a monad comprises the operator  $T$  itself as well as “let” and “val”, just as a group is not only the underlying set but also the multiplication operation. Again, just as groups the monads admit an equational axiomatisation. Of course, additional type and term formers depending on the particular instance are needed. In our case this is the type **ref** and the operations for reading and writing.

$$\begin{aligned}
\mathbf{read} : \mathbf{ref} &\rightarrow T(\mathbf{int}) \\
\mathbf{write} : \mathbf{ref} &\rightarrow \mathbf{int} \rightarrow T(\mathbf{unit})
\end{aligned}$$

To model exceptions one would use  $T(A) = A \cup \mathbf{exn}^+$  where **exn** is a set modelling a basic type of exceptions. One then introduces constants

$$\text{throw} : \text{exn} \rightarrow T(A)$$

$$\text{catch} : (\text{exn} \times T(A)) \rightarrow T(A)$$

Function types in the metalanguage are always pure thus do a priori not have side effects. The function space in the concrete language is then rendered as  $A \rightarrow T(B)$ .

The pure programming language Haskell allows one to define one's own monads and boasts syntactic abbreviations which create the illusion of working in a side-effecting concrete language where in fact one always works with a pure meta language.

The type structure of the meta language is somewhat richer than that of the concrete language in that the latter has no pendant of types like  $\text{int} \rightarrow \text{int}$  which denote (in the meta language!) side-effect-free functions. More on this topic can be found in [3].

## 6. Effects

Our goal is to employ *refined* types to gain information about the nature of side effects possibly occurring during the evaluation of a term *statically* that is to say without knowing the environment in which the term is to be evaluated.

For each location  $\ell \in \mathbb{L}$  we introduce the type  $\text{ref}_\ell$  which contains precisely that one reference: a *singleton type*. Furthermore, for each  $\ell \in \mathbb{L}$  we introduce the two *elementary effects*  $rd_\ell$  (reading) and  $wr_\ell$  (writing). An *effect* is then a set of elementary effects.

The refined types are given by the grammar

$$A, B := \text{unit} \mid \text{int} \mid \text{bool} \mid \text{ref}_\ell \mid A \times B \mid A \xrightarrow{\varepsilon} B$$

where  $\ell \in \mathbb{L}$  and  $\varepsilon$  is an effect. The *refined typing judgement* takes the form

$$\Gamma \vdash e : A, \varepsilon$$

where now  $\Gamma$  maps variables to refined types. The refined typing judgement says that the evaluation of  $e$  in an environment that respects the refined typing in  $\Gamma$  will yield a result in  $A$  and moreover exhibit at most the side effects declared in  $\varepsilon$ . Later on we will provide rigorous definitions of “respects” and “exhibit” allowing us to justify effect-dependent program transformations.

The effect in  $A \xrightarrow{\varepsilon} B$  is often called a *latent effect* for it will be brought to bear not immediately but only once a function of that type is being evaluated.

We abbreviate  $A \xrightarrow{\emptyset} B$  by  $A \rightarrow B$  and we abbreviate  $\Gamma \vdash e : A, \emptyset$  by  $\Gamma \vdash e : A$ . Also, we may elide the empty context.

For example, for arbitrary  $\ell \in \mathbb{L}$  we have the following refined typings.

$$\begin{array}{c}
\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma(x) = \mathbf{ref}_\ell \quad \Gamma(y) = \mathbf{int}}{\Gamma \vdash x := y : \mathbf{unit}, \{wr_\ell\}} \quad \frac{\Gamma(x) = \mathbf{ref}_\ell}{\Gamma \vdash !x : \mathbf{int}, \{rd_\ell\}} \\
\frac{\Gamma \vdash e_1 : A_1, \varepsilon_1 \quad \Gamma, x:A_1 \vdash e_2 : A_2, \varepsilon_2}{\Gamma \vdash \mathbf{let } x \leftarrow e_1 \mathbf{ in } e_2 : A_2, \varepsilon_1 \cup \varepsilon_2} \quad \frac{\Gamma(x) = A_1 \xrightarrow{\varepsilon} A_2 \quad \Gamma(y) = A_1}{\Gamma \vdash x y : A_2, \varepsilon} \\
\frac{\Gamma \vdash e_1 : A, \varepsilon_1 \quad \Gamma \vdash e_2 : A, \varepsilon_2 \quad \Gamma(x) = \mathbf{bool}}{\Gamma \vdash \mathbf{if } x \mathbf{ then } e_1 \mathbf{ else } e_2 : A, \varepsilon_1 \cup \varepsilon_2} \quad \frac{\Gamma \vdash x y : A_2, \varepsilon}{\Gamma, x:A \vdash e : B, \varepsilon} \\
\frac{\Gamma \vdash \mathbf{if } x \mathbf{ then } e_1 \mathbf{ else } e_2 : A, \varepsilon_1 \cup \varepsilon_2 \quad \Gamma \vdash \lambda x. e : A \xrightarrow{\varepsilon} B, \emptyset}{\Gamma \vdash e : A_1, \varepsilon_1 \quad A_1 <: A_2 \quad \varepsilon_1 \subseteq \varepsilon_2} \\
\frac{\Gamma \vdash e : A_2, \varepsilon_2}{A_1 <: A_2 \quad A_3 <: A_4 \quad \varepsilon_1 \subseteq \varepsilon_2} \\
\frac{A <: A}{A_2 \xrightarrow{\varepsilon_1} A_3 <: A_1 \xrightarrow{\varepsilon_2} A_4}
\end{array}$$

Figure 4. Typing rules for effect typing

$$\begin{array}{l}
\vdash \ell := 9 : \mathbf{unit}, \{wr_\ell\} \\
\vdash \ell := !\ell : \mathbf{unit}, \{rd_\ell, wr_\ell\} \\
\vdash \lambda x. \ell := x : \mathbf{int} \xrightarrow{\{wr_\ell\}} \mathbf{unit} \\
\vdash \mathbf{COUNTER} : \mathbf{ref}_\ell \rightarrow (\mathbf{unit} \xrightarrow{\{rd_\ell, wr_\ell\}} \mathbf{unit}) \times \\
\quad (\mathbf{unit} \xrightarrow{\{rd_\ell\}} \mathbf{int}) \times (\mathbf{unit} \xrightarrow{\{wr_\ell\}} \mathbf{unit}) \\
\vdash \mathbf{MEMO} : \mathbf{ref}_{\ell_1} \rightarrow \mathbf{ref}_{\ell_2} \rightarrow (\mathbf{int} \xrightarrow{\varepsilon_1} \mathbf{int}) \xrightarrow{\varepsilon_2} \mathbf{int} \xrightarrow{\varepsilon_3} \mathbf{int}
\end{array}$$

In the last typing  $\varepsilon_1$  is arbitrary,  $\varepsilon_2 = \varepsilon_1 \cup \{wr_{\ell_1}, wr_{\ell_2}\}$  and  $\varepsilon_3 = \varepsilon_1 \cup \{rd_{\ell_1}, wr_{\ell_1}, rd_{\ell_2}, wr_{\ell_2}\}$ . This last example shows that effect annotation can only conservatively approximate the actual behaviour of a program. A computation  $\mathbf{MEMO} \ell_1 \ell_2 f v$  may or may not exhibit the effect  $\varepsilon_1$  depending on whether or not the required  $f$ -value can be looked up or not. The second example is also interesting in that, semantically, the term in question is side-effect free, but our type system has no way of discovering this. However, we could use our semantics to justify a stronger type system that would ascribe a pure typing to the term  $\ell := !\ell$ .

## 7. Effect system

We now give typing rules that formally define the refined typing judgment. We assume ANF, i.e., the typing rules apply to terms in ANF which will save a considerable amount of repetition. Recall that  $\Gamma \vdash A$  abbreviates  $\Gamma \vdash A, \emptyset$ , etc. Also some trivial rules, e.g., for arithmetic operators are omitted. The rules are in Figure 4. The last three rules in Figure 4 define and use an auxiliary subtyping relation that allows one to weaken the refined typing, so as to obtain common refined typings e.g. in two branches of a conditional. For example, without subtyping we would not be able to assign a type to  $\mathbf{if } x \mathbf{ then } \lambda y. y \mathbf{ else } \lambda y. !\ell$ .

*Effect polymorphism and type inference* In general, any given term will have infinitely many different types that are mutually incomparable in the subtyping relation. The information about a term that can be gleaned through the typing rules thus comprises an infinite set. In order to analyse programs automatically

one will thus use a finitary notation for such infinite families of types. It is common to employ type schemes like in the Hindley-Milner type inference for the simply-typed lambda calculus and used in the ML programming language.

These type schemes then contain type variables, effect variables, and location variables. In addition type schemes may contain constraints stipulating inclusion of certain effects and difference of locations.

The types for the running examples given above can be considered as instances of such type schemes if one interprets the metavariables  $\varepsilon, \ell$ , etc. as actual variables.

Just as in the case of ML type inference one will allow schematic types for let-bound variables which can then be instantiated in different ways in the body. Thus, the following term cannot be typed with the rules given above but can be typed with the help of type schemes:

$$\mathbf{let} \ f \Leftarrow \lambda x. x := 0 \ \mathbf{in} \ (f \ \ell_1); (f \ \ell_2)$$

with  $\ell_1$  and  $\ell_2$  two different locations.

Using type schemes one would assign the type  $\mathbf{ref}_\ell \xrightarrow{wr_\ell} \mathbf{unit}$  which can be instantiated with  $\ell = \ell_1$  and  $\ell = \ell_2$ . This is particularly important in the case of recursive definitions “let rec” where, incidentally, ML does not allow multiple instantiations in the body. For any given term there then exists a most general schematic type which can be efficiently computed with an appropriate extension of the Hindley-Milner inference algorithm. We will not consider the area of automatic type inference in these notes; for more information see [9,14,24] which are the standard references on effect typing.

## 8. Effect-based program transformation

We will now employ effect information in order to justify program transformations. Here is a first example of such a program transformation.

*Dead code elimination* Suppose that a term contains a subterm  $e$  which admits the typing  $\Gamma \vdash e : \mathbf{unit}, \varepsilon$  in its context where  $\{\ell \mid wr_\ell \in \varepsilon\} = \emptyset$  then  $e$  can be replaced by  $()$ , i.e., removed. In what sense is such replacement admissible? In any case the semantics of the two terms are in general not equal. We shall answer that question in the next section.

To facilitate the formulation of further equivalences we introduce the following notations:

$$\mathbf{rds}(\varepsilon) = \{\ell \mid rd_\ell \in \varepsilon\}$$

$$\mathbf{wrs}(\varepsilon) = \{\ell \mid wr_\ell \in \varepsilon\}$$

Here are further transformations:

*Code motion:* Suppose that two terms  $e_1, e_2$  admit the following typings in their context:  $\Gamma \vdash e_1 : A, \varepsilon_1$  and  $\Gamma \vdash e_2 : A, \varepsilon_2$  where  $\text{rds}(\varepsilon_1) \cap \text{wrs}(\varepsilon_2) = \emptyset$ ,  $\text{wrs}(\varepsilon_1) \cap \text{rds}(\varepsilon_2) = \emptyset$ , and  $\text{wrs}(\varepsilon_1) \cap \text{wrs}(\varepsilon_2) = \emptyset$ . Then the term  $(e_1, e_2)$  (i.e.,  $\text{let } x_1 \leftarrow e_1 \text{ in let } x_2 \leftarrow e_2 \text{ in } (x_1, x_2)$  in ANF) may be replaced with  $\text{let } x_2 \leftarrow e_2 \text{ in let } x_1 \leftarrow e_1 \text{ in } (x_1, x_2)$ . Along with the general rule that semantically equal terms may be replaced by one another this means that the order of evaluation of  $e_1$  and  $e_2$  may be exchanged.

*Duplicated code* Suppose that the term  $e$  can be typed as  $\Gamma \vdash e : A, \varepsilon$  in its context where  $\text{rds}(\varepsilon) \cap \text{wrs}(\varepsilon) = \emptyset$ . Then  $(e, e)$  may be replaced with  $\text{let } x \leftarrow e \text{ in } (x, x)$ . This then means that the duplicate execution of  $e$  can be contracted to a single one.

*Pure Lambda Hoist* Suppose that a term  $e$  can be typed as  $\Gamma \vdash e : A$  in its context; thus,  $e$  is *pure* (free of side effects). Then the following two terms can be replaced by one another:

$$\begin{aligned} & \lambda x. \text{let } y \leftarrow e \text{ in } e'(x, y) \\ & \text{let } x \leftarrow e \text{ in } \lambda y. e'(x, y) \end{aligned}$$

This means that side-effect-free computations that do not depend on formal parameters can be extracted (“hoisted”) from a function (method) body and evaluated once and for all in advance.

We note that in general the typing assumptions made in the context are crucial for the required typings to hold. Here is an example of this situation:

$$f : \text{unit} \xrightarrow{\varepsilon} A \vdash f(\ell:=1) : A, \varepsilon \cup \{wr_\ell\}$$

Thus *Duplicated Computation* applies to the term  $f(\ell:=1)$  provided that  $\text{rds}(\varepsilon) \cap (\text{wrs}(\varepsilon) \cup \{\ell\}) = \emptyset$ . Thus, the effect typing allows one to add information about side effects to the specification of a function (method), here  $f$ , and to use that information at the call sites of the function for the purposes of program transformation.

Note that the semantics of  $\text{let } x \leftarrow f(\ell:=1) \text{ in } (x, x)$  and  $(f(\ell:=1), f(\ell:=1))$  are different because the semantics does not model effect annotations. Refining the semantics so that it does allow to capture such information is the main goal of this work as are extensions to a richer language of course.

Before, however, refining the semantics, we must make it clear in what sense we want program equivalences to hold.

## 9. Observational equivalence

We choose to justify program equivalences as observational equivalence, i.e., we will show that two terms deemed equivalent can be replaced by one another within any closed term of basic type. Formally,

**Definition 9.1.** Let  $v_1$  and  $v_2$  be closed and pure terms of some refined type  $A$ , i.e.,

$\vdash v_1 : A$

$\vdash v_2 : A$

Then  $v_1$  and  $v_2$  are *observationally equivalent at type  $A$* , written,

$$\models v_1 \equiv v_2 : A$$

if for all closed and pure terms  $v : A \xrightarrow{\varepsilon} \mathbf{bool}$  with  $\varepsilon$  arbitrary one has that whenever

$$\llbracket v \ v_1 \rrbracket (s_0) = (s_1, b_1)$$

$$\llbracket v \ v_2 \rrbracket (s_0) = (s_2, b_2)$$

then  $b_1 = b_2$ . Here  $s_0$  is the initial state defined for example by  $s_0(\ell) = 0$  for all  $\ell$ .

The term  $v$  thus represents the observation made about  $v_1$  and  $v_2$ . The state reached after the observation is discarded, only the boolean result is retained.

However, in our situation the following is the case: if  $\models v_1 = v_2 : A$  and  $\vdash v : A \xrightarrow{\varepsilon} \mathbf{bool}$  and

$$\llbracket v \ v_1 \rrbracket (s_0) = (s_1, b_1)$$

$$\llbracket v \ v_2 \rrbracket (s_0) = (s_2, b_2)$$

then  $s_1 = s_2$  follows, too. To see this, fix  $\ell \in \mathbb{L}$  and suppose that  $s_1(\ell) = c$ . Define another observation  $v' := \lambda x.v \ x; (!\ell == c)$  where  $==$  is an equality test. From the observational equivalence of  $v_1$  and  $v_2$  applied to  $v'$  it can then be concluded that  $s_2(\ell) = c$ , too. Note that this would not be so if local references not visible from the outside are admitted.

By a similar argument it follows that observational equivalence would stay the same if observations of integer type or multiple observations were allowed.

**Definition 9.2.** Two terms  $e_1, e_2$ , where  $x_1:A_1, \dots, x_n:A_n \vdash e_1, e_2 : A, \varepsilon$  are observationally equivalent at  $A, \varepsilon$  if  $\lambda x_1 \dots \lambda x_n. \lambda y. e_i$  for  $i = 1, 2$  are observationally equivalent at  $A_1 \rightarrow \dots \rightarrow A_n \rightarrow \mathbf{unit} \xrightarrow{\varepsilon} A$ .

Notice that observational equivalence is always type-dependent.

**Proposition 9.1.** *If two terms have equal semantics then they are observationally equivalent at any type. The converse is in general not true.*

This is because the definition of observational equivalence refers to terms only via their semantics.

## 10. Relational semantics

Before we embark on the formal definition let us informally motivate the concept of modelling effects as sets of relations to be preserved. Assume that we have only

two locations  $X$  and  $Y$ , i.e.  $\mathbb{L} = \{X, Y\}$  and let  $c : \mathbb{S} \rightarrow \mathbb{S}$  be (a denotation of) a command. How can we formalise that  $c$

- may read  $X$  but not  $Y$
- may write  $Y$  but not  $X$

Note that  $c$ , being a denotation, we do not have access to its execution trace. One obvious attempt at giving such a formalisation would be to require that there exists a function  $f : \mathbb{Z} \rightarrow \mathbb{Z}$  such that

$$\begin{aligned} c(s).X &= s.X \\ c(s).Y &= s.Y \vee c(s).Y = f(s.X) \end{aligned}$$

and the decision in the disjunction only depends on  $s.X$

In order to formalise the latter restriction we might additionally require the existence of a function  $g : \mathbb{Z} \rightarrow \{0, 1\}$  such that

$$c(s).Y = g(s.X) \cdot s.Y + (1 - g(s.X)) \cdot f(s.X)$$

Let us temporarily say that  $c$  has property *Direct* if such functions  $f, g$  exist.

Alternatively, we can formalise the intended effect property by requiring that for all relations  $R \subseteq \mathbb{S} \times \mathbb{S}$  *compatible with these effects* we have

$$\forall s, s'. sRs' \Rightarrow c(s)Rc(s')$$

where  $R$  is “compatible with these effects” if

- whenever  $sRs'$  then  $s.X = s'.X$  (because we may read  $X$ )
- whenever  $sRs'$  and  $v \in \mathbb{Z}$  then  $s[Y \mapsto v]Rs'[Y \mapsto v]$  (because we may write  $Y$ )

Let us temporarily say that command  $c$  has property *Relational* if this is the case. One can now show that the two properties *Direct* and *Relational* are equivalent. The property *Direct* has the advantage of being (perhaps) more intuitive; the formulation *Relational*, on the other hand, eases compositional reasoning; e.g. it is straightforward that if  $c$  satisfies *Relational* so does the composition  $c; c$ . Furthermore, *Relational* is easy to generalise to more complicated sets of effects: A command may exhibit some effect  $\varepsilon$  if it preserves all store relations that are compatible with that effect; the more effects are exhibited the fewer relations are to be preserved.

The relational semantics of effects has the additional advantage that it generalises to a symmetric, transitive relation in the case where commands return values with nontrivial observational equivalence.

We will now formalise this notion and extend it to all types. This will result in a refined relational semantics that distinguishes refined types with equal underlying type and in particular validates the above program equivalences. Of course, this semantics should not be coarser than observational equivalence; as we shall see this will be a consequence of the relational semantics being compositionally defined and nontrivial (not all values receive the same denotation).

**Definition 10.1.** A binary relation  $R$  on a set  $A$  is a *partial equivalence relation* (PER) if  $R$  is symmetric and transitive. The *support* of  $R$  is the set  $\text{supp}(R) = \{x \mid xRx\}$ . The restriction of  $R$  to  $\text{supp}(R)$  is an equivalence relation. Thus, a PER corresponds to a partition of a subset of  $A$  into classes.

For each refined type  $A$  we define the underlying simple type  $|A|$  in the obvious way by removing all effect information. This notation also applies to typing contexts and judgements.

It is our goal to define a PER  $\llbracket A \rrbracket$  on  $\llbracket |A| \rrbracket$  for each type  $A$  such that the support  $\text{supp}(\llbracket A \rrbracket)$  singles out those elements of  $\llbracket |A| \rrbracket$  that respect the effect information contained in  $A$ . On this support, the equivalence relation  $\llbracket A \rrbracket$  should refine observational equivalence and identify at least those elements whose equivalence can be deduced from the equational theory generated by our program equivalences and congruence rules.

As already mentioned in the introduction we describe effects of computations by sets of state relations to be preserved.

**Definition 10.2.** For each effect  $\varepsilon$  we define a set of relations  $\mathcal{R}_\varepsilon$  on states as follows:

$$\begin{aligned} R \in \mathcal{R}_\emptyset &\iff R \subseteq \mathbb{S} \times \mathbb{S} \\ R \in \mathcal{R}_{\varepsilon_1 \cup \varepsilon_2} &\iff R \in \mathcal{R}_{\varepsilon_i} \text{ for } i = 1, 2 \\ R \in \mathcal{R}_{rd_\ell} &\iff \forall s \ s' . sRs' \Rightarrow s.\ell = s'.\ell \\ R \in \mathcal{R}_{wr_\ell} &\iff \forall s \ s' \ v . sRs' \Rightarrow s[\ell \mapsto v]Rs'[\ell \mapsto v] \end{aligned}$$

**Definition 10.3 (relational semantics).** The definition of the relational semantics is then given as follows.

$$\begin{aligned} (v, v') \in \llbracket A \rrbracket &\iff v = v' \text{ and } A \in \{\mathbf{bool}, \mathbf{unit}, \mathbf{int}\} \\ (v, v') \in \llbracket \mathbf{ref}_\ell \rrbracket &\iff v = v' = \ell \\ ((v_1, v_2), (v'_1, v'_2)) \in \llbracket A_1 \times A_2 \rrbracket &\iff (v_i, v'_i) \in \llbracket A_i \rrbracket \text{ for } i = 1, 2 \\ (f, f') \in \llbracket A \xrightarrow{\varepsilon} B \rrbracket &\iff \forall (v, v') \in \llbracket A \rrbracket . (f \ v, f' \ v') \in T_\varepsilon(\llbracket B \rrbracket) \\ (f, f') \in T_\varepsilon(A) &\iff \forall R \in \mathcal{R}_\varepsilon . \forall s, s' . sRs' \Rightarrow s_1Rs'_1 \wedge vAv' \\ &\text{where } f(s) = (s_1, v), f'(s') = (s'_1, v') \end{aligned}$$

Consider that  $f \in T(\llbracket \mathbf{int} \rrbracket)$  and that  $(f, f) \in T_{\{rd_\ell\}}(\llbracket \mathbf{int} \rrbracket)$ . Let  $R \in \mathcal{R}_{\{rd_\ell\}}$  be given by

$$sRs' \iff s(\ell) = s'(\ell)$$

We then find that if  $sRs'$  and  $f(s) = (s_1, v), f(s') = (s'_1, v')$  then in particular  $v = v'$  so the result depends only on  $\ell$ . Now fix  $s$  and consider the relation  $R' \in \mathcal{R}_{\{rd_\ell\}}$  given by

$$s' R' s'' \iff s' = s'' = s$$

Thus, if  $f(s) = s_1, v$  then  $s_1 R s_1$  so  $s_1 = s$ , i.e.,  $f$  did indeed not write.

By experimenting a bit more one sees that the relational semantics does indeed capture our intuitions about reading and writing. For example, if  $(f, f) \in T_{\{rd_\ell, wr_{\ell'}\}}$  then we can show by similar arguments that  $f$  will modify at most the  $\ell'$  component of the store and if it does so then to a value that depends only on  $\ell$ , etc.

We can now state and prove the fundamental lemma which essentially asserts the soundness of our effect typing rules with respect to our relational semantics. More precisely, it says that the effect information obtained syntactically does indeed adequately describe the semantic behaviour of the term. The proof of the fundamental lemma is by induction on typing derivations and does not present any surprises.

**Theorem 10.1 (fundamental lemma).** *Suppose that  $\Gamma \vdash e : A, \varepsilon$  and  $(\eta(x), \eta'(x)) \in \llbracket \Gamma(x) \rrbracket$  for all  $x \in \text{dom}(\Gamma)$ . Then  $(\llbracket e \rrbracket \eta, \llbracket e \rrbracket \eta') \in T_\varepsilon(\llbracket A \rrbracket)$ .*

**Theorem 10.2 (observational equivalence).** *Let  $e, e'$  be terms with  $\Gamma \vdash e : A, \varepsilon$  and  $\Gamma \vdash e' : A, \varepsilon$ . For all  $\eta, \eta'$  with  $(\eta(x), \eta'(x)) : \llbracket \Gamma(x) \rrbracket$  for  $x \in \text{dom}(\Gamma)$  suppose that  $(\llbracket e \rrbracket \eta, \llbracket e \rrbracket \eta') \in T_\varepsilon(\llbracket A \rrbracket)$ .*

This is proved by applying the fundamental lemma to the observation and using the fact that the relational semantics at base types is equality.

**Theorem 10.3 (Program equivalences).** *The abovementioned program equivalence “dead code” is semantically valid in the following sense: If  $\Gamma \vdash e : \mathbf{unit}, \varepsilon$  and  $\text{wrs}(\varepsilon) = \emptyset$  and  $(\eta(x), \eta'(x)) : \llbracket \Gamma(x) \rrbracket$  for all  $x \in \text{dom}(\Gamma)$  then  $(\llbracket e \rrbracket \eta, \llbracket () \rrbracket \eta') \in T_\varepsilon(\llbracket \mathbf{unit} \rrbracket)$ . Analogous statements hold for the other equivalences, “code motion”, “duplicated code”, “pure lambda hoist”.*

More generally: if the equivalence of two terms  $e, e'$  where  $\Gamma \vdash e : A, \varepsilon$  and  $\Gamma \vdash e' : A, \varepsilon$  is derivable using equational reasoning (with reflexivity and congruence rules restricted to well-typed terms) from the four program equivalences, and universally valid semantic equivalences (terms with equal denotational semantics ( $\llbracket - \rrbracket$ ) are equivalent) then for  $\eta, \eta'$  as above it holds that  $(\llbracket e \rrbracket \eta, \llbracket e' \rrbracket \eta') \in T_\varepsilon(\llbracket A \rrbracket)$ .

## 11. Dynamic allocation

We add a new basic type **ref** and a new term former **ref**( $-$ ) such that **ref**( $e$ ) : **ref** when  $e : \mathbf{int}$ . The idea is that **ref**( $e$ ) generates a new reference initialised with the value of the variable  $e$ .

This constructs permits more elegant versions of our example programs:

$COUNTER \stackrel{def}{=} \text{let } x \leftarrow \text{ref}(0) \text{ in } (\lambda u. x := !x + 1, \lambda u. !x, \lambda u. x := 0) :$   
 $(\text{unit} \rightarrow \text{unit}) \times (\text{unit} \rightarrow \text{int}) \times (\text{unit} \rightarrow \text{unit})$

$MEMO \stackrel{def}{=} \text{let } l_1 \leftarrow \text{ref}(0) \text{ in let } l_2 \leftarrow \text{ref}(0) \text{ in } \lambda f.$   
 $l_1 := 0; l_2 := 0;$   
 $\lambda x. \text{if } x = !l_1 \text{ then } !l_1 \text{ else}$   
 $\text{let } u \leftarrow f \ x \text{ in } l_1 := x; l_2 := u; u :$   
 $(\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$

In order to model this semantically, we assume a set  $\mathbb{S}$  of states endowed with the following operations: There is a constant  $\emptyset \in \mathbb{S}$ , the empty state. If  $s \in \mathbb{S}$  then  $\text{dom}(s) \subseteq \mathbb{L}$  and if  $\ell \in \text{dom}(s)$  then  $s.\ell \in \mathbb{Z}$  is a value; if  $v \in \mathbb{Z}, \ell \in \text{dom}(s)$  then  $s[\ell \mapsto v] \in \mathbb{S}$ ; finally  $\text{new}(s, v)$  yields a pair  $(\ell, s')$  where  $\ell \in \mathbb{L}$  and  $s' \in \mathbb{S}$ . These operations are subject to the following axioms:

$$\begin{aligned}
&\text{dom}(\emptyset) = \emptyset \\
&\text{dom}(s[\ell \mapsto v]) = \text{dom}(s) \\
&(s[\ell \mapsto v]).\ell' = \text{if } \ell = \ell' \text{ then } v \text{ else } s.\ell' \\
&\text{new}(s, v) = (\ell, s') \Rightarrow \text{dom}(s') = \text{dom}(s) \cup \{\ell\} \wedge \\
&\quad \ell \notin \text{dom}(s) \wedge s'.\ell = v
\end{aligned}$$

This abstract datatype can be implemented in a number of ways, e.g., as finite maps. We do not want to commit ourselves to any particular implementation, in particular, we do not make the perhaps plausible assumption that the newly allocated reference  $\ell_0$  when  $\text{new}(s, v) = (s', \ell_0)$  depends only on  $\text{dom}(s)$ .

Assuming ANF we then put

$$\begin{aligned}
&\llbracket !x \rrbracket \eta \ s = s.\eta(x) \\
&\llbracket x := y \rrbracket \eta \ s = s[\eta(x) \mapsto \eta(y)] \\
&\llbracket \text{ref}(x) \rrbracket \eta \ s = \text{new}(s, \eta(x))
\end{aligned}$$

The other semantic equations remain unchanged.

## 12. Refined typing with regions

We will now extend refined typing to dynamic allocation. To that end we partition the allocated memory area into disjoint regions. These regions are not reflected physically at runtime; they merely play a role in the type system. Thus, the denotational semantics is not affected by the regions in any way.

We assume an infinite supply  $\text{Regs}$  of region (identifiers) and define refined types as follows:

$$A, B := \text{unit} \mid \text{int} \mid \text{bool} \mid \text{ref}_r \mid A \times B \mid A \xrightarrow{\varepsilon} B$$

where an effect  $\varepsilon$  is a subset of the set of *elementary effects*

$$\{rd_r, wr_r, al_r \mid r \in Regs\}$$

Thus, accesses to individual references are approximated by accesses to regions. Furthermore, a new elementary effect  $al_r$  signalling an allocation within region  $r$  is introduced. The refined type  $\mathbf{ref}_r$  represents the set of locations within region  $r$ .

The typing rules are analogous to the global case. For example, we can deduce:

$$COUNTER : (\mathbf{unit} \xrightarrow{\{rd_r, wr_r\}} \mathbf{unit}) \times (\mathbf{unit} \xrightarrow{\{rd_r\}} \mathbf{int}) \times (\mathbf{unit} \xrightarrow{\{wr_r\}} \mathbf{unit})$$

where  $r$  is an arbitrary region. We give explicitly the rules for allocation and for writing:

$$\frac{\Gamma(x) = \mathbf{int}}{\Gamma \vdash \mathbf{ref}(x) : \mathbf{ref}_r : \{al_r\}} \quad \frac{\Gamma(x) = \mathbf{ref}_r \quad \Gamma(y) = \mathbf{int}}{\Gamma \vdash x := y : \mathbf{unit}, \{wr_r\}}$$

Note that the choice of region  $r$  in the rule for allocation is arbitrary. Of course, when typing a program, we will try to use as many regions as possible so as to maximise the applicability of program transformations which will for example require that simultaneous write accesses happen in different regions.

With integer references only it happens rarely that we are not able to spend a different region on every single allocation made. Once we have structured data like lists and recursion, even primitive recursion, it will no longer be possible to do so.

An altogether new feature is the following *masking rule*

$$\frac{\Gamma \vdash e : A, \varepsilon \quad r \text{ does not occur in } \Gamma \text{ or } \tau}{\Gamma \vdash e : A, \varepsilon \setminus \{wr_r, rd_r, al_r\}}$$

This rule allows one to delete effects concerning a region  $r$  that is mentioned neither in the types of the free variables nor in the type of the result. If, for instance,  $e$  is a closed expression of type  $\mathbf{int}$  which internally uses one or more instances of *COUNTER*, then  $e$  can a priori be typed as

$$\vdash e : \mathbf{int}, \{rd_r, wr_r, al_r\}$$

with  $r$  an arbitrary region. The masking rule then allows us to derive the pure typing  $\vdash e : \mathbf{int}$ .

We will now extend the relational semantics to this situation and in particular show that effect-dependent program equivalences continue to hold in the presence of masking. Thus, the above term  $e$  could be evaluated several times instead of once, etc. without altering the semantic meaning in the sense of observational equivalence.

We remark that this region-based type system was originally introduced by Tofte and Talpin in order to enable block-structured memory management without garbage collection. The idea is to type a program (automatically) in the region type system with aggressive use of masking. At runtime masked regions are then deallocated (freed) after the masked expression has been evaluated. We remark that unlike in our application this does require a certain amount of runtime

support in the form of a table that records which physical locations belong to which region. In our example the *COUNTER* objects would be deallocated once the result has been computed.

Tofte and Talpin employ this kind of memory management in their ML-Kit compiler. More recently, region-based memory management has appeared (for obvious reasons) in Real-Time Java ([www.rtsj.org](http://www.rtsj.org)).

### 12.1. Formal preliminaries

**Definition 12.1 (Quasi PER).** A binary relation  $R$  on a set  $A$  is a *Quasi PER* (QPER) if whenever  $xRy$  and  $zRy$  and  $zRw$ , then  $xRw$ , too. In other words,  $RR^{-1}R = R$ .

QPERs have been introduced as “difunctional relations” in a 1940s paper and appear from time to time in the literature. The terminology QPER is non-standard.

If  $R$  is a QPER define  $R_1 = RR^{-1}$  and  $R_2 = R^{-1}R$ . Both  $R_1$  and  $R_2$  are PERs and  $R$  defines a bijection between their respective sets of equivalence classes.

**Definition 12.2.** If  $R$  is any binary relation on a set  $A$  we denote  $QPER(R)$  the least QPER containing  $R$ .

The following characterisation of  $QPER(R)$  is interesting but will not be needed in the sequel.

**Proposition 12.1.** *Let  $R$  be a binary relation on a set  $A$ . One has*

$$(x, x') \in QPER(R) \iff \forall k, k' : A \Rightarrow \{0, 1\}. (\forall y, y'. yRy' \Rightarrow ky = k'y') \Rightarrow kx = k'x'$$

In the previous sections we approximated contextual equivalence by a partial equivalence relation  $\llbracket A \rrbracket$  for each refined type  $A$ . In the presence of dynamic allocation such a simple-minded setup will no longer work and we move to families of relations (known as “Kripke logical relation”) indexed by state 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.

Furthermore, the relations  $\llbracket A \rrbracket_\varphi$  for  $\varphi$  a parameter will not be PERs but only QPERs.

**Lemma 12.2.** *Let  $R, S$  be binary relations on sets  $A, B$ , let  $Q$  be a QPER on  $C$  and  $f, f' : A \times B \rightarrow C$  be functions. If*

$$\forall a, a', b, b'. aRa' \wedge bSb' \Rightarrow f(a, b)Qf'(a', b')$$

*then*

$$\forall a, a', b, b'. aQPER(R)a' \wedge bQPER(S)b' \Rightarrow f(a, b)Qf'(a', b')$$

*Proof.* Fix  $a, a'$  such that  $aRa'$  and form  $U = \{(b, b') \mid f(a, b)Qf'(a', b')\}$ .

The assumption yields  $S \subseteq U$  so, since  $U$  is a QPER (!), we obtain

$$\forall a, a', b, b'. aRa' \wedge bQPER(S)b' \Rightarrow f(a, b)Qf'(a', b')$$

The claim now follows using the QPER  $\{(a, a') \mid \forall b, b'. bQPER(S)b' \Rightarrow f(a, b)Qf'(a', b')\}$ .  $\square$

### 12.2. Partial bijections

The relational interpretation of types will depend on parameters that approximate the current store layout. The central ingredient of parameters are partial bijections which we now define. They describe the locations which belong to a given region in two computations that are deemed equivalent.

**Definition 12.3 (partial bijection).** A *partial bijection* is a triple  $(L, L', f)$  where  $L, L'$  are finite subsets of  $\mathbb{L}$  and  $f \subseteq L \times L'$  such that  $(l_1, l'_1) \in f$  and  $(l_2, l'_2) \in f$  imply  $l_1 = l_2 \Leftrightarrow l'_1 = l'_2$ .

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

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

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

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

### 12.3. Parameters

When modelling global store we approximated observational equivalence by a partial equivalence relation  $\llbracket A \rrbracket$  for each refined type  $A$ . In the presence of dynamic allocation such a simple-minded setup will no longer work and we move to families of relations 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. Aspect (a) of parameters is expressed by assigning to each region identifier a partial bijection between locations in the store.

For aspect (b) of parameters we introduce a special symbol  $\tau \notin \text{Regs}$  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 12.4 (parameter).** A *parameter*  $\varphi$  is a function that assigns to every region  $r$  (including the silent region) a partial bijection  $\varphi(r)$  such that

- distinct regions map to disjoint partial bijections.
- $\text{dom}(\varphi) = \bigcup_{r \in \text{Regs} \cup \{\tau\}} \text{dom}(\varphi(r))$  and  $\text{dom}'(\varphi) = \bigcup_{r \in \text{Regs} \cup \{\tau\}} \text{dom}'(\varphi(r))$  are both finite sets so that in fact  $\varphi(r) = \emptyset$  for all but finitely many regions  $r$ .

If  $\varphi$  and  $\varphi'$  are parameters such that

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

then  $\varphi, \varphi'$  are called *disjoint* and we write  $\varphi \otimes \varphi'$  for the obvious juxtaposition of  $\varphi$  and  $\varphi'$  given by  $(\varphi \otimes \varphi')(r) = \varphi(r) \otimes \varphi'(r)$ .

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

The set of parameters is partially ordered by  $\varphi' \geq \varphi \iff \varphi' = \varphi \otimes \psi$  for some necessarily unique  $\psi$ .

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

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

We let  $\varphi - r$  denote the parameter defined by

$$\begin{aligned} (\varphi - r)(r') &= \varphi(r') \text{ when } r' \neq r \\ (\varphi - r)(\tau) &= \varphi(\tau) \otimes (\text{dom}(\varphi(r)), \text{dom}'(\varphi(r)), \emptyset) \end{aligned}$$

**Definition 12.5 (state relations).** If  $L, L'$  are sets of locations, a *state relation* on  $L, L'$  is defined as a nonempty relation  $R \subseteq \mathbb{S} \times \mathbb{S}$  such that whenever  $(s, s') \in R$  and  $s \sim_L s_1$  and  $s' \sim_{L'} s'_1$  then  $(s_1, s'_1) \in R$ , too. We write  $\text{StRel}(L, L')$  for the set of all state relations on  $L, L'$ .

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

**Definition 12.6 (relations and effects).** Let  $R$  be a state relation on  $\text{dom}(\varphi), \text{dom}'(\varphi)$ . We say that  $R$  *respects*  $\varepsilon$  at  $\varphi$  if it is preserved by all commands that exhibit only  $\varepsilon$  on the state layout delineated by  $\varphi$ . Formally, we define:

- $R$  respects  $\{rd_r\}$  at  $\varphi$  if  $(s, s') \in R$  implies  $s.l = s'.l'$  for all  $(\ell, \ell') \in \varphi(r)$ ;
- $R$  respects  $\{wr_r\}$  at  $\varphi$  if for all  $(s, s') \in R$  and for all  $(\ell, \ell') \in \varphi(r)$  and  $v \in \mathbb{Z}$ , we have  $(s[\ell \mapsto v], s'[\ell' \mapsto v]) \in R$ ;
- $R$  respects  $\{al_r\}$  always.

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

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

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

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

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

We also define the following:

$$s \sim_{\varphi} s' \iff \forall r \in \text{Reqs}. \forall (\ell, \ell') \in \varphi(r). s.\ell = s'.\ell'$$

### 13. Logical Relation

This section defines the relational semantics of refined types.

**Definition 13.1 (logical relation).** Let  $A$  be a refined type and  $\varphi$  be a parameter. We define a QPER  $\llbracket A \rrbracket_{\varphi}$  on  $\llbracket |A| \rrbracket$  by the following clauses.

$$\begin{aligned} \llbracket A \rrbracket_{\varphi} &\equiv \{(v, v) \mid v \in \llbracket |A| \rrbracket\} \text{ when } A \in \{\text{int}, \text{bool}, \text{unit}\} \\ \llbracket \text{ref}_r \rrbracket_{\varphi} &\equiv \varphi(r) \\ \llbracket A \times B \rrbracket_{\varphi} &\equiv \llbracket A \rrbracket_{\varphi} \times \llbracket B \rrbracket_{\varphi} \\ \llbracket A \xrightarrow{\varepsilon} B \rrbracket_{\varphi} &\equiv \{(f, f') \mid \forall \varphi' \geq \varphi. \forall (x, x') \in \llbracket A \rrbracket_{\varphi'}. \\ &\quad (f(x), f'(x')) \in (T_{\varepsilon} \llbracket B \rrbracket)_{\varphi'}\} \\ (T_{\varepsilon} Q)_{\varphi} &\equiv \text{QPER}(\{(f, f') \mid s, s' \models \varphi \Rightarrow \\ &\quad \forall R \in \mathcal{R}_{\varepsilon}(\varphi). s R s' \Rightarrow s_1 R s'_1 \wedge \\ &\quad \exists \psi. (\psi(r) \neq \emptyset \Rightarrow r \in \text{als}(\varepsilon)) \wedge s_1, s'_1 \models \varphi \otimes \psi \wedge \\ &\quad s_1 \sim_{\psi} s'_1 \wedge (v, v') \in Q_{\varphi \otimes \psi} \\ &\quad \text{where } (s_1, v) = f s \text{ and } (s'_1, v') = f' s'\}) \end{aligned}$$

We define  $\llbracket \Theta \rrbracket_{\varphi}$  by  $\llbracket \Theta \rrbracket_{\varphi} \equiv \{(\gamma, \gamma') \mid \forall i. (\gamma(x_i), \gamma'(x_i)) \in \llbracket A_i \rrbracket_{\varphi}\}$  where  $\Theta = x_1 : A_1, \dots, x_n : A_n$ .

The definition of the logical relation on computation types deserves some explanation. First, it says that the store behaviour of two related computations must respect all relations that are compatible with the declared effect. Since these relations are completely unconstrained on the silent region  $\tau$ , this implies in particular that the silent region may neither be read nor modified. The existential quantifier asserts a (disjoint) extension  $\psi$  of the current parameter  $\varphi$  which is to hold all newly allocated references. The result values  $(v, v')$  are then required to be related with respect to the extended parameter  $\varphi \otimes \psi$ . Note that if  $v$  and  $v'$  contain newly allocated references then  $(v, v') \in \llbracket B \rrbracket_{\varphi}$  will in general not hold.

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

**Lemma 13.1 (Monotonicity).** *If  $\varphi' \geq \varphi$  then  $\llbracket A \rrbracket_{\varphi'} \supseteq \llbracket A \rrbracket_{\varphi}$ .*

**Lemma 13.2 (QPER).** *For each  $\varphi$  the relation  $\llbracket A \rrbracket_{\varphi}$  is a QPER.*

**Lemma 13.3 (masking).** *Suppose that  $r$  does not occur anywhere in  $A$ . Then  $\llbracket A \rrbracket_{\varphi} = \llbracket A \rrbracket_{\varphi-r}$ .*

**Lemma 13.4 (extension).**

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

The following establishes semantic soundness for our subtyping relation.

**Lemma 13.5 (Soundness of subtyping).** *If  $A_1 \leq A_2$  then for all  $\varphi$  one has  $\llbracket A_1 \rrbracket_{\varphi} \subseteq \llbracket A_2 \rrbracket_{\varphi}$ .*

We now have the following ‘fundamental theorem’ of logical relations, which states that terms are related to themselves.

**Theorem 13.6 (Fundamental Theorem).** *If  $\Gamma \vdash e : A, \varepsilon$  and  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_{\varphi}$ , then*

$$(\llbracket e \rrbracket_{\gamma}, \llbracket e \rrbracket_{\gamma'}) \in T_{\varepsilon}(\llbracket A \rrbracket)_{\varphi}.$$

*Proof.* The proof is by induction on typing derivations; thanks to Lemma 12.2 we can do as if the  $QPER(-)$  closures were absent in applications of the induction hypothesis. We now give the cases for the typing rule for “let” and for the masking rule.

*Case “let”:* Here  $e$  is  $\text{let } x \leftarrow e_1 \text{ in } e_2$  and we have  $\Gamma \vdash e_1 : A_1, \varepsilon_1$  and  $\Gamma, x:A_1 \vdash e_2 : A : \varepsilon_2$ , so  $e = \text{let } x \leftarrow e_1 \text{ in } e_2$  and  $\varepsilon = \varepsilon_1 \cup \varepsilon_2$ .

Let  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_{\varphi}$  and  $R \in \mathcal{R}_{\varepsilon}(\varphi)$  and  $sRs'$  and  $s, s' \models \varphi$  and  $(\check{s}_1, \check{v}) = \llbracket e_1 \rrbracket_{\gamma} s$  and  $(\check{s}'_1, \check{v}') = \llbracket e_1 \rrbracket_{\gamma'} s'$ . By the induction hypothesis applied to  $e_1$  and the aforementioned use of Lemma 12.2 we have  $s_1Rs'_1$  and  $\check{s}_1, \check{s}'_1 \models \varphi \otimes \check{\psi}$  and  $(\check{v}_1, \check{v}'_1) \in \llbracket A_1 \rrbracket_{\varphi \otimes \check{\psi}}$ .

By monotonicity we have  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_{\varphi \otimes \check{\psi}}$  and so  $(\gamma[x \mapsto \check{v}], \gamma'[x \mapsto \check{v}']) \in \llbracket \Gamma, x:A_1 \rrbracket_{\varphi \otimes \check{\psi}}$ . We conclude with the induction hypothesis applied to  $e_2$ .

*Case “masking rule”:* Suppose  $\Gamma \vdash e : A, \varepsilon$  and  $r \notin \Gamma, A$ . Suppose  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_{\varphi_0}$ . Put  $f = \llbracket e \rrbracket_{\gamma}, f' = \llbracket e \rrbracket_{\gamma'}, \varphi = \varphi_0 - r$ . By the masking lemma we have  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_{\varphi}$ . Let us apply IH( $e$ ) to  $\varphi \otimes [r \mapsto \emptyset]$ . We obtain  $(f, f') \in T_{\varepsilon}(\llbracket A \rrbracket)_{\varphi \otimes [r \mapsto \emptyset]}$ .

We should now prove  $(f, f') \in T_{\varepsilon-r}(\llbracket A \rrbracket)_{\varphi}$  whence  $(f, f') \in T_{\varepsilon-r}(\llbracket A \rrbracket)_{\varphi_0}$  by masking lemma again.

So assume  $s, s' \models \varphi$  and  $R \in \mathcal{R}_{\varepsilon-r}(\varphi)$  and  $sRs'$ . By the extension lemma we have  $R \in \mathcal{R}_{\varepsilon}(\varphi \otimes [r \mapsto \emptyset])$  so the induction hypothesis gives  $s_1Rs'_1$  and  $\psi$  such that  $s_1, s'_1 \models \varphi \otimes [r \mapsto \emptyset] \otimes \psi$  and  $(v, v') \in \llbracket A \rrbracket_{\varphi \otimes [r \mapsto \emptyset] \otimes \psi}$  where  $f s = (s_1, v)$  and  $f' s' = (s'_1, v')$ .

The masking lemma gives  $(v, v') \in \llbracket A \rrbracket_{\varphi \otimes (\psi-r)}$  and we are done.  $\square$

## 14. Applications

We introduce the notation

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

It expresses that  $s$  and  $s'$  agree on those locations that are read given effect  $\varepsilon$ .

We also define

$$\begin{aligned} \text{nwr}_{s_\varphi}(\varepsilon) &= \text{dom}(\varphi) \setminus \bigcup_{r \in \text{wrs}(\varepsilon)} \text{dom}(\varphi(r)) \\ \text{nwr}'_{s'_\varphi}(\varepsilon) &= \text{dom}'(\varphi) \setminus \bigcup_{r \in \text{wrs}(\varepsilon)} \text{dom}'(\varphi(r)) \end{aligned}$$

Thus  $\text{nwr}_{s_\varphi}(\varepsilon)$  and  $\text{nwr}'_{s'_\varphi}(\varepsilon)$  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 14.1.** *Suppose  $\Gamma \vdash e : T_\varepsilon A$  and  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_\varphi$  and  $s_0, s'_0 \models \varphi$  and  $\llbracket e \rrbracket \gamma s_0 = (s_1, x)$  and  $\llbracket e \rrbracket \gamma' s'_0 = (s'_1, x')$ .*

*If  $s_0 \sim_{\text{rds}_\varphi(\varepsilon)} s'_0$  then there exists  $\psi$  with  $\psi(r) \neq \emptyset \Rightarrow r \in \text{als}_\varepsilon$  and disjoint from  $s_0, s'_0$  such that*

1.  $s_1, s'_1 \models \varphi \otimes \psi$  and  $(x, x') \in \llbracket A \rrbracket_{\varphi \otimes \psi}$  and  $s_1 \sim_\psi s'_1$ .
2.  $s_0 \sim_{\text{nwr}_{s_\varphi}(\varepsilon)} s_1$  and  $s'_0 \sim_{\text{nwr}'_{s'_\varphi}(\varepsilon)} s'_1$ .
3. For each  $(\ell, \ell') \in \varphi(r)$  where  $r \in \text{Regs}$  we have either
  - $s_0.\ell = s_1.\ell$  and  $s'_0.\ell' = s'_1.\ell'$  (unchanged) or
  - $s_1.\ell = s'_1.\ell'$  (identically written).
4. Suppose that  $\ell \in \text{dom}(\varphi)$  but there is no  $\ell', r$  such that  $(\ell, \ell') \in \varphi(r)$ . Then  $s_0.\ell = s_1.\ell$ . A symmetric statement holds for  $s'_0, s'_1$ .

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

**Definition 14.1 (semantic equality).** Suppose that  $\Gamma \vdash e_i : A, \varepsilon$  for  $i = 1, 2$ . We write  $\Gamma \models e_1 = e_2 : A, \varepsilon$  to mean that for all  $\varphi$  and  $(\gamma, \gamma') \in \llbracket \Theta \rrbracket_\varphi$  one has  $(\llbracket e_1 \rrbracket \gamma, \llbracket e_2 \rrbracket \gamma') \in \llbracket A \rrbracket_\varphi$ .

**Proposition 14.2.** *If  $\Gamma \models e_1 = e_2 : A, \varepsilon$  then  $e_1$  and  $e_2$  are observationally equivalent.*

**Proposition 14.3.** *If  $\Gamma \models e_1 = e_2 : A, \varepsilon$  and  $\Gamma \models e_2 = e_3 : A, \varepsilon$  then  $\Gamma \models e_1 = e_3 : A, \varepsilon$ .*

*Proof.* Suppose that  $(\gamma, \gamma') \in \llbracket \Gamma \rrbracket_\varphi$ . We should prove  $(\llbracket e_1 \rrbracket \gamma, \llbracket e_3 \rrbracket \gamma') \in T_\varepsilon(\llbracket A \rrbracket)_\varphi$ .

We have  $(\llbracket e_i \rrbracket \gamma, \llbracket e_i \rrbracket \gamma') \in T_\varepsilon(\llbracket A \rrbracket)_\varphi$  for  $i = 1, 2, 3$  by the Fundamental Lemma. The assumption gives  $(\llbracket e_i \rrbracket \gamma, \llbracket e_{i+1} \rrbracket \gamma') \in T_\varepsilon(\llbracket A \rrbracket)_\varphi$  for  $i = 1, 2$ . We conclude by QPER-ness.  $\square$

Similarly, we can show that semantic equality is symmetric and transitive on well-typed terms and that it is a congruence with respect to all term formers.

We can now state our program equivalences in the form of semantic equalities.

**Proposition 14.4 (duplicated computation).** *Suppose that  $\Gamma \vdash e : A, \varepsilon$  and suppose that  $\text{rds}(\varepsilon) \cap \text{wrs}(\varepsilon) = \text{als}(\varepsilon) = \emptyset$ . Thus,  $e$  reads and writes on disjoint portions of the store and makes no allocations except possibly in the silent region. Then  $\Gamma \models e_1 = e_2 : A, \varepsilon$  where*

$$\begin{aligned} e_1 &:= \text{let } x \leftarrow e \text{ in val } (x, x) \\ e_2 &:= \text{let } x \leftarrow e \text{ in let } y \leftarrow e \text{ in val } (x, y) \end{aligned}$$

Analogous statements hold for dead code, pure lambda hoist, commuting computations.

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

**Proposition 14.5.** *Let  $C$  be a cartesian closed category and  $T$  be a strong monad on  $C$ . Suppose that in the Kleisli category  $C_T$  the laws of dead computation, commuting computations, duplicated computations, lambda hoist are valid. Then  $C_T$  is cartesian closed.*

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

## 15. Recursion

In order to model recursion we associate a Scott domain (in fact  $\omega$ -chain complete partial order with bottom would suffice) with every type and in particular define  $\llbracket A \xrightarrow{\varepsilon} B \rrbracket$  as the domain of strict, continuous maps from  $\llbracket A \rrbracket$  to  $T\llbracket B \rrbracket$  where  $TX$  is the domain of strict maps from the flat domain of states to  $\mathbb{S} \otimes X$  where  $\otimes$  denotes strict product. I.e., a computation either does not terminate (is bottom) or returns a pair consisting of a new state and a (non-bottom) value. Recursive definitions can then be interpreted in the usual way as suprema of Kleene chains.

One must then make sure that all semantic relations  $\llbracket A \rrbracket_\varphi$  are admissible QPERS, ie contain  $(\perp, \perp)$  and are closed under suprema of chains in the sense that if  $(v_i, v'_i) \in \llbracket A \rrbracket_\varphi$  then  $(\bigsqcup_i v_i, \bigsqcup_i v'_i) \in \llbracket A \rrbracket_\varphi$ , too. This is achieved by adding appropriate clauses at base types and adapting the closure operator  $QPER(R)$  so as to yield the least admissible QPER comprising  $R$ .

Unfortunately, the program equivalence “dead code” breaks down in the presence of recursions since the elided code fragment might not terminate. One can fix this by introducing a termination condition as a semantic side condition to be discharged either by semantic reasoning or by some other type system, e.g. by introducing a “nontermination effect”. Similar considerations apply to pure lambda hoist.

## 16. Conclusion

We have given a relational semantics to a region-based effect type system for a higher-order language with dynamically allocated store. The relational semantics

is shown sound for contextual equivalence and thus provides a powerful proof principle for the latter. We have used the semantics to establish the soundness of a collection of useful effect-based program transformations. It would probably be very hard to establish these directly from the definition of contextual equivalence and no such proof appears to exist in the literature.

There has been a great deal of previous work on the soundness of region-based memory management and of its close cousin, encapsulated monadic state, as provided by `runST` in Haskell [12]. We mention some particularly relevant references. Banerjee et al. [2] translate the region calculus into a variant of System F and give a denotational model showing that references in masked regions do not affect the rest of the computation. Moggi and Sabry [18] prove syntactic type soundness for encapsulated lazy state. Fluet and Morrisett [8] bring the two lines of work together by giving a type- and behaviour-preserving translation from a variant of the region calculus into an extension of System F with a region-indexed family of monads. Naumann [19] uses simulation relations to capture a notion of observational purity for boolean-valued specifications that allows mutation of encapsulated state.

The general problem of modelling and proving equivalences in languages with dynamically allocated store and higher order features is a difficult one, with a very long history [25]. 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 [11,15,20,21,22,6]. 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 [15]. 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 [16], bisimulation was adapted for the untyped lambda calculus by Abramsky [1]. Other researchers, particularly Sumii and Pierce, subsequently developed notions of bisimulation for typed lambda calculi that could deal with the kind of encapsulation (data abstraction) given by existential types [23]. These methods have recently been refined by Koutavas and Wand, and applied to an untyped higher-order language with storage [13] 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.

Like the characterisation of contextual equivalence with logical relations given by Pitts and Stark, this does not directly solve the problem at hand here. Firstly, we use *typed* contextual equivalence which has fewer observing contexts and is thus coarser than untyped contextual equivalence. Indeed, in [13] an extension to the typed case is left as an open question.

Secondly, as already mentioned, our applications concern *relative* contextual equivalences involving unknown programs. It is not yet clear how the bisimulation method fares with those. It is, however, true that it could be interesting and profitable to base our technical development on bisimulations rather than logical relations.

## References

- [1] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Directions in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, 1988.
- [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, J. Hughes, and E. Moggi. Monads and effects. In *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS*. Springer-Verlag, 2002.
- [4] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation.
- [5] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Reading, writing, and relations: Towards extensional semantics for effect analyses. In *4th Asian Symposium on Programming Languages and Systems (APLAS)*, LNCS, 2006.
- [6] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *LNCS*, 2005.
- [7] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation*. ACM, 1993.
- [8] M. Fluet and G. Morrisett. Monadic regions. *Journal of Functional Programming*, 2006. to appear.
- [9] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, Massachusetts, August 1986.
- [10] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [11] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, 1984.
- [12] S. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4), 1995.
- [13] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, pages 141–152, 2006.
- [14] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM Symposium on Principles of Programming Languages (POPL)*, 1988.
- [15] A. R. Meyer and K. Sieber. Towards a fully abstract semantics for local variables: Preliminary report. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1988.
- [16] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
- [17] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asilomar, CA*, pages 14–23, 1989.

- [18] E. Moggi and A. Sabry. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming*, 11(6), 2001.
- [19] D. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, To appear.
- [20] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.
- [21] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.
- [22] 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.
- [23] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2005.
- [24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), June 1984. Revised from LICS 1992.
- [25] R. D. Tennent and D. R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.