Operational Reasoning for Functions with Local State

Andrew Pitts and Ian Stark

Abstract

Languages such as ML or Lisp permit the use of recursively defined function expressions with locally declared storage locations. Although this can be very convenient from a programming point of view it severely complicates the properties of program equivalence even for relatively simple fragments of such languages—such as the simply typed fragment of Standard ML with integer-valued references considered here. This paper presents a method for reasoning about *contextual equivalence* of programs involving this combination of functional and procedural features. The method is based upon the use of a certain kind of *logical relation* parameterised by relations between program states. The form of this logical relation is novel, in as much as it involves relations not only between program expressions, but also between program continuations (also known as *evaluation contexts*). The authors found this approach necessary in order to establish the 'Fundamental Property of logical relations' in the presence of both dynamically allocated local state and recursion. The logical relation characterises contextual equivalence and yields a proof of the best known context lemma for this kind of language—the Mason-Talcott 'ciu' theorem. Moreover, it is shown that the method can prove examples where such a context lemma is not much help and which involve representation independence, higher order memoising functions, and profiling functions.

Contents

1 Introduction

Lisp and ML are *functional* programming languages because they treat functions as values on a par with more concrete forms of data: functions can be passed as arguments, can be returned as the result of computation, can be recursively defined, and so on. They are also *procedural* languages because they permit the use of references (or 'cells', or 'locations') for storing values: references can be created dynamically and their contents read and updated as expressions are evaluated. This paper presents a method for reasoning about the equivalence of programs involving this combination of functional and procedural features. What emerges is an operationally-based form of reasoning about functions with local state that seems to be both intuitive and theoretically powerful. Throughout we assume a passing familiarity with the language Standard ML (Milner, Tofte, and Harper 1990) and its associated terminology. If in difficulty, see (Paulson 1991).

Some motivation

The combination of functional and procedural features in Standard ML is very expressive. For example, it permits the programmer to exploit the modularity of the functional idiom (Hughes 1989) in defining high-level control structures for manipulating program state. The combination is also useful from the point of view of efficiency, since the use of local storage permits the efficient implementation of some functions and data structures with purely functional observable behaviour. As a simple example, consider the following 'memoised' version of the factorial function in Standard ML.

Example 1.1.

$$
\begin{array}{l}
\mathtt{val\,f} \;=\; \mathtt{let\,val\,a} = \mathtt{ref\,0\,and\,r} = \mathtt{ref\,1} \qquad (dec_1)\\
\qquad\qquad \mathtt{fun\,f'\,x} = (\mathtt{if\,x} = \mathtt{0\,then\,1\,else\,x} * \mathtt{f'}(\mathtt{x}-\mathtt{1}))\\
\qquad\quad \mathtt{in}\\
\qquad\qquad \mathtt{fn\,x} \Rightarrow ((\mathtt{if\,x} = \mathtt{!a\,then}\,()\,\mathtt{else}\,(\mathtt{a} := \mathtt{x}; \mathtt{r} := \mathtt{f'\,x})); \mathtt{!r})\\
\qquad\quad \mathtt{end}
\end{array}
$$

The local references `a` and `r` are used to store the argument and the result of the most recent invocation of the function; and the function acts like the purely functional factorial

$$\mathtt{fun\,f\,x} \;=\; (\mathtt{if\,x} = \mathtt{0\,then\,1\,else\,x} * \mathtt{f}(\mathtt{x}-\mathtt{1})) \qquad (dec_2)$$

except that when called with the same argument twice in succession it uses the cached result `!r`, saving recomputation. So evaluating

$$\mathtt{let}\; dec_1 \;\mathtt{in}\; (\mathtt{f\;1000} + \mathtt{f\;1000})\; \mathtt{end}$$

will yield the same integer result as evaluating

$$\texttt{let}\ dec_2\ \texttt{in}\ (\texttt{f}\ 1000 + \texttt{f}\ 1000)\ \texttt{end}$$

but will only involve computing the factorial of 1000 once. Of course in this simple example a similar saving can easily be achieved without leaving the purely functional part of the language, for example with

$$\texttt{let}\ dec_2\ ;\ \texttt{val x} = (\texttt{f}\ 1000)\ \texttt{in}\ (\texttt{x} + \texttt{x})\ \texttt{end}.$$

The point is that in general such functional transformations may require complicated static analysis of the context, whereas the transformation involving the memoised version is simply one of replacing occurrences of dec_2 with dec_1 uniformly in any context. The correctness of this optimisation amounts to the assertion that dec_1 and dec_2 are contextually equivalent. In general one says that two phrases e_1 and e_2 in a programming language are *contextually equivalent*, and writes $e_1 \cong e_2$, if for all contexts $C[-]$, *i.e.* for all expressions which become complete programs when the hole '$-$' is filled with e_1 or e_2, executing the program $C[e_1]$ yields exactly the same observable results as executing $C[e_2]$.

Why are dec_1 and dec_2 contextually equivalent? While it may be easy to see for some particular context such as $\texttt{let}\ [-]\ \texttt{in}\ (\texttt{f}\ 1000 + \texttt{f}\ 1000)$, that $C[dec_1]$ and $C[dec_2]$ evaluate to the same result, it is quite a different matter to prove that this is so for *all* contexts. Nevertheless there are reasons for believing that $dec_1 \cong dec_2$ holds, based upon the 'privacy' of locally declared references and properties of the state that remain invariant during evaluation. Here is how the argument goes.

Informal 'proof' of $dec_1 \cong dec_2$. Consider the following property:

$$\text{the integer stored in } \texttt{r} \text{ is the factorial of that stored in } \texttt{a}. \qquad (1.1)$$

Note that if dec_1 is evaluated then the two references created satisfy (1.1) (since 1 is the factorial of 0). Moreover, the function value bound to `f` as a result of that evaluation is such that if (1.1) holds before evaluating an application $\texttt{f}(\texttt{n})$, then it continues to hold afterwards and the value returned agrees with what we would have obtained using dec_2 instead of dec_1 (namely the factorial of `n`).

Given any context $C[-]$, since `a` and `r` are bound identifiers in dec_1 and evaluation of expressions respects α-conversion, we may assume that `a` and `r` do not occur in $C[-]$; so, *the only way that the contents of the created references* `a` *and* `r` *could be mutated during evaluation of* $C[dec_1]$ *is through applications of the function value bound to* `f`. It follows from the previous paragraph both that the property (1.1) is invariant throughout the evaluation of $C[dec_1]$ and that any result returned is the same as for $C[dec_2]$. Since $C[-]$ was arbitrary, dec_1 and dec_2 are contextually equivalent. □?

The reason why this is only an informal proof resides mainly in the statement in italics about how a context can make use of its 'hole', which certainly needs

further justification. To emphasise this point, we tease the reader with a similar informal 'proof' of contextual equivalence that turns out to be false. Recall that `int ref` is the Standard ML type of integer references: its values are addresses of integer storage locations, and those values can be tested for equality—meaning equality of the reference addresses rather than equality of their contents. Consider the following declarations of functions of type `int ref` $\to$ `int ref`.

Example 1.2.

$$\begin{array}{lll} \texttt{val f} = \texttt{let val a} = \texttt{ref 0 and b} = \texttt{ref 0} & & (dec_3) \\ \quad \texttt{in} \\ \quad\quad \texttt{fn c} \Rightarrow (\texttt{if c} = \texttt{a then b else a}) \\ \quad \texttt{end} \\ \texttt{val f} = \texttt{let val a} = \texttt{ref 0 and b} = \texttt{ref 0} & & (dec_4) \\ \quad \texttt{in} \\ \quad\quad \texttt{fn c} \Rightarrow (\texttt{if c} = \texttt{b then b else a}) \\ \quad \texttt{end} \end{array}$$

False 'proof' of $dec_3 \cong dec_4$. Given any context $C[-]$, since `a` and `b` are bound identifiers in dec_i ($i = 3, 4$) and evaluation of expressions respects α-conversion, we may assume that `a` and `b` do not occur in $C[-]$. Thus in evaluating $C[dec_i]$ *any value of type* `int ref` *which is supplied by* $C[-]$ *to the function declared by* dec_i *cannot be either* `a` *or* `b`. Therefore any such application will always use the second branch of the conditional and return `a`. So evaluation of $C[dec_3]$ will produce the same result as evaluating $C[dec_4]$. Since $C[-]$ was arbitrary, dec_3 and dec_4 are contextually equivalent. □?

The italicised part of this 'proof' is of the same kind as in the previous case, but this time it is false. Indeed dec_3 and dec_4 are not contextually equivalent. As with any contextual inequivalence, this can be demonstrated rigorously by exhibiting a context $C[-]$ for which $C[dec_3]$ and $C[dec_4]$ produce different results. Such a context is

$$\texttt{let } [-]\texttt{; val c} = \texttt{ref 0 in } (\texttt{f}(\texttt{f c}) = \texttt{f c}) \texttt{ end}$$

since in this case $C[dec_3]$ evaluates to `false` whereas $C[dec_4]$ evaluates to `true`. For in the environment created by evaluating the local declarations in $C[dec_3]$ (respectively $C[dec_4]$), `f c` evaluates to `a` (respectively `a`), hence `f`$($`f c`$)$ evaluates to `b` (respectively `a`) and therefore the test `f`$($`f c`$) =$ `f c` yields `false` (respectively `true`). Note that contrary to the expectation in the false 'proof' given above, even though $C[-]$ does not know about the local references `a` and `b`, it is able to feed them as arguments to `f` after one application of `f` to some external reference `c`.

Incidentally, Example 1.2 demonstrates that the following naïve *extensionality principle* fails for Standard ML functions:

> Two expressions F and F' of function type $\sigma \to \sigma'$ are contextually equivalent if for all values V of type σ, $F\,V$ and $F'\,V$ are contextually equivalent expressions of type σ'.

We have just seen that if F_3 and F_4 are the expressions of type $\texttt{int ref} \to \texttt{int ref}$ that occur on the right-hand sides of the declarations dec_3 and dec_4 respectively, then $F_3 \not\cong F_4$. On the other hand, one can show for all values $\texttt{c} : \texttt{int ref}$ that $F_3\,\texttt{c}$ and $F_4\,\texttt{c}$ are contextually equivalent. So F_3 and F_4 do not satisfy the above extensionality principle.

This failure of extensionality is not merely a result of mixing higher order functions with imperative features. For in Reynolds' Idealised Algol (1981, 1982) with its call-by-name function application and restriction of local state just to commands (*i.e.* expressions of type `unit`, in ML terminology), such an extensionality principle *does* hold: see (Pitts 1997). Rather, it is the fact that in ML access to local references can be passed out of their original scope during evaluation which complicates the properties of contextual equivalence. We saw this when demonstrating $dec_3 \not\cong dec_4$ in Example 1.2. Incidentally, it is worth noting that although this example makes use of equality testing on references, the failure of extensionality in ML does not depend upon this feature. (Indeed, in the fragment of ML we use in this paper this equality test is definable from more primitive ones: see Remark 2.1.) The operationally-based parametric logical relation we present in this paper provides a characterisation of contextual equivalence that yields a rigorous underpinning for the kind of informal argument used in Example 1.1, while avoiding the pitfalls illustrated by Example 1.2.

Some background

The methods presented here for reasoning about recursive functions and local storage are rooted in the work of O'Hearn and Tennent (1995) and Seiber (1995). These authors use relational parametricity (Reynolds 1983) and logical relations (Plotkin, 1973, 1980) to give denotational models of Algol-like languages which match the operational behaviour of local variables better than previous models did. Since our goal is not to produce 'fully abstract' models, but rather to identify practically useful proof methods for contextual equivalence, there is some advantage to concentrating on operationally-based versions of these relational techniques. This was done for Algol-like languages in (Pitts 1997). Here we carry out a similar program for an ML-like language. For the reasons given in the previous subsection, the difficulties which have to be overcome to produce useful proof methods for ML contextual equivalence are greater than those for Algol. Nevertheless, we obtain a fairly light-weight tool compared with the mathematical structures involved in the denotational models, and one which relates directly to the syntax and structural operational semantics of the language. But of course these operationally-based techniques would not have arisen without the previous, denotational insights. Furthermore, the method we use

to establish the fundamental properties of the operationally-based logical relation with respect to recursive program constructs relies on operational analogues of familiar denotational methods (*viz.* fixed point induction and admissibility properties).

Mason and Talcott have developed a number of operational methods for reasoning about Lisp programs with destructive update (Mason and Talcott 1991a; Mason and Talcott 1992a; Honsell, Mason, Smith, and Talcott 1995). Like us, they highlight the issue of functions with local state, which they call 'objects' (1991b, 1992b). Notions of 'constraint' and 'equivalence under constraints' are used, which can be loosely identified with the use we make in this paper of relations between states. These lead to a set of reasoning principles that match certain aspects of our operational logical relation. Their **(inv)** expresses the fact that properties of local store are preserved; while their **(abstract)** and **(abstractable)** say that if two functions preserve some property of store, and whenever it holds they give the same result, then they are equivalent. Proofs based on these principles are similar in some ways to those given in Section 5. There are however limitations to these methods, which our work removes. For example, the validity of **(inv)**, **(abstract)** and **(abstractable)** is restricted to first-order functions over atoms, as a consequence of their 'hands-on' proof through direct consideration of reduction in certain contexts. Our logical relation has no such restriction (witness the higher order profiler of Example 5.8). Our techniques then can be seen as a certain generalisation to higher types of the results of Mason *et al*, through the powerful machinery of parameterised logical relations.

Overview of the paper

In Section 2 we introduce a language of Recursive Functions with local State, called ReFS, which is the vehicle for the formal development in the rest of this paper. Syntactically, it is a simply typed lambda calculus: there are ground types for booleans, integers, the unit value, and integer references; higher types are built up over these using product and function type constructors. We give the structural operational semantics of ReFS in terms of an inductively defined evaluation relation

$$s, M \Downarrow V, s' \ . \tag{1.2}$$

This and the associated definition of contextual equivalence are quite standard, and make ReFS equivalent to a fragment of SML according to its definition in (Milner, Tofte, and Harper 1990). Harper and Stone (1996) reformulate the operational semantics of SML in terms of transitions between configurations containing a component for the current program continuation, or evaluation context. (See also Harper, Duba, and MacQueen 1993.) The advantage of this approach is that it can give a *structurally inductive* characterisation of the termination predicate, $\exists V, s' \, (s, M \ \Downarrow \ V, s')$ used to define contextual equivalence. Accordingly, in

Section 3 we introduce a new termination relation

$$\langle s, K, M\rangle\downarrow \tag{1.3}$$

where the component K formalises the ReFS evaluation contexts. The relation (1.3) is defined by induction on the structure of M and K, and contains the original termination relation for $\Downarrow$ as a retract. We are able to exploit the structural nature of our formulation of termination to streamline the induction proofs that arise when proving properties of contextual equivalence. A case in point is the proof of the Unwinding Theorem 3.2 that completes this section. It expresses a compactness property of recursive function values with respect to termination which we need later to prove a crucial preservation property of our parametric logical relation (Proposition 4.8(xv)).

The logical relation itself is introduced in Section 4 and its fundamental properties established. It is parameterised by binary relations between states. Apart from being operationally- rather than denotationally-based, we are able to make a pleasing simplification of (O'Hearn and Tennent 1993, Section 6), in that our parameters are just *arbitrary* (non-empty) state-relations without any extra structure of a partial bijection on the underlying address names. In fact the definition of the logical relation is rather different from previous such definitions for languages with local state, because it involves binary relations between evaluation contexts K as well as binary relations between expressions M.

We found this approach unavoidable in order to establish the Fundamental Property of the logical relation (Theorem 4.9) and hence its connection with contextual equivalence (Theorem 4.10). The reason has to do with the interaction between recursion and the fact that the 'size' of the state (measured by the number of storage locations allocated) may grow in a non-trivial fashion during evaluation in a language like ReFS. Thus in (1.2), the number of locations in the final state s' may be strictly greater than the number in the initial state s and we cannot 'garbage collect' that part of s' involving these extra locations, because the value V may be a function closure using those locations. Now in defining a logical relation parameterised by state-relations and based upon the evaluation relation (1.2), it is natural to use existential quantification over relations on the dynamically created part of s': this is what the authors did for their nu-calculus in (Pitts and Stark 1993), for example. However, such an existential quantification destroys the (operational analogue of the) admissibility property needed to show that recursive program constructs respect the logical relation—without which there would be no connection between contextual equivalence and the logical relation.[1]

By contrast, the logical relation we give here takes account of evaluation contexts rather than final states and uses the termination relation (1.3), which makes no explicit mention of final states. This allows us to avoid any use of existential quantification over state relations in the definition and renders the proof

[1] This problem did not surface in (Pitts and Stark 1993) because the nu-calculus does not contain any recursive features.

of the Fundamental Property relatively straightforward. The price we pay is that the definition of the logical relation between expressions is intertwined with the definition of a 'dual' relation between evaluation contexts (program continuations). However, it is a price worth paying, since not only does it allow us to prove the crucial Fundamental Property of the logical relation, but also we are able to characterise contextual equivalence in terms of the logical relation[2] and deduce the Mason-Talcott 'closed instantiations of uses' theorem for ReFS as a corollary (see Theorem 4.10). Moreover, we show in Section 5 that we can recover a technique for proving ReFS contextual equivalence involving existential quantification over 'locally invariant' state relations which is reminiscent of the methods of (Pitts and Stark 1993; Pitts 1997). This Principle of Local Invariants (Proposition 5.1) is put to work in Section 5 to prove examples of contextual equivalence involving the notion of representation independence, higher-order memoising functions, and higher-order profiling functions. We also examine the limitations of this method, giving an example (Example 5.9) of two contextually equivalent ReFS expressions that are not easily seen to be logically related.

In the final Section 6 we discuss some desirable extensions of the ReFS language and how our techniques might be extended to cope with them.

2 Recursive functions with local state

The examples discussed in the Introduction involved the interaction between recursively declared functions and dynamically created, mutable references for storing integer values. They were phrased in a simply typed fragment of Standard ML with ground types `bool` (booleans), `int` (integers), `unit` (one-element type), and `int ref` (integer storage locations). In this section we introduce a typed lambda calculus called ReFS—a language of Recursive Functions with local State. It is essentially equivalent to the fragment of Standard ML we have in mind and will be the vehicle for the formal development in the rest of this paper.

The ReFS language

The syntax of ReFS is given in Figure 1. It takes an unusually reduced form, in that most operators may take only values as arguments. This is essentially a technical convenience: it means that all the sequential aspects of the language devolve onto the let construct, and can therefore be treated uniformly. Unrestricted forms are easily defined in terms of let as shown in Figure 2, and we shall use them freely. Note that the ReFS let is much simpler than that of ML, being neither recursive nor polymorphic.

ReFS has two kinds of identifier, variables ($x, y, f, g, \ldots$) and location constants ($\ell, \ell', \ldots$). The latter occur explicitly in ReFS expressions because we prefer to

[2] A similar characterisation for the nu-calculus definitely fails for the logical relation of (Pitts and Stark 1993).

$$
\begin{array}{rl}
\textit{Expressions} & M ::= V \mid \textit{if } V \textit{ then } M \textit{ else } M \mid V \textit{ op } V \mid \textit{fst}(V) \mid \textit{snd}(V) \\
& \quad \mid \textit{ref}(V) \mid !V \mid V := V \mid VV \mid \textit{let } x = M \textit{ in } M \\
\textit{Values} & V ::= x \mid \textit{true} \mid \textit{false} \mid n \mid () \mid \ell \mid \textit{rec}\, x(x).\, M \mid (V, V) \\
\textit{Types} & \sigma ::= \textit{bool} \mid \textit{int} \mid \textit{unit} \mid \textit{loc} \mid \sigma \to \sigma \mid \sigma \times \sigma
\end{array}
$$

where

$x \in \mathit{Var}$ an infinite set of *variables*,
$\ell \in \mathit{Loc}$ an infinite set of *locations*,
$n \in \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ the set of integers,
$\mathit{op} \in \{+, -, =, \leq, \ldots\}$ a finite set of arithmetic operations and relations.

Figure 1: ReFS syntax

$$
\textit{if } M_1 \textit{ then } M_2 \textit{ else } M_3 \stackrel{\text{def}}{=} \textit{let } x = M_1 \textit{ in } (\textit{if } x \textit{ then } M_2 \textit{ else } M_3) \quad \text{where } x \notin \mathit{fv}(M_2, M_3)
$$

and similar clauses for $M_1 \textit{ op } M_2$, (M_1, M_2), $\textit{fst}(M)$, $\textit{snd}(M)$, $M_1 M_2$, $\textit{ref}(M)$, $!M$, and $M_1 := M_2$.

$$
\begin{array}{rcl}
\lambda\, x.\, M & \stackrel{\text{def}}{=} & \textit{rec}\, f(x).\, M \quad \text{where } f \notin \mathit{fv}(M) \\
M; M' & \stackrel{\text{def}}{=} & \textit{let } x = M \textit{ in } M' \quad \text{where } x \notin \mathit{fv}(M') \\
Y_{cbv} & \stackrel{\text{def}}{=} & \textit{rec}\, y(f).\, \lambda\, x.\, f(yf)x \\
\textit{while } B \textit{ do } M & \stackrel{\text{def}}{=} & (\textit{rec}\, f(x).\, \textit{if } B \textit{ then } (M; f()) \textit{ else } ())() \quad \text{where } f, x \notin \mathit{fv}(B, M) \\
\textit{let } x = M_1; x' = M_2 \textit{ in } M_3 & \stackrel{\text{def}}{=} & \textit{let } x = M_1 \textit{ in } (\textit{let } x' = M_2 \textit{ in } M_3) \\
\textit{let } (x_1, x_2) = M_1 \textit{ in } M_2 & \stackrel{\text{def}}{=} & \textit{let } x = M_1 \textit{ in let } x_1 = \textit{fst}(x) \textit{ in} \\
& & \quad \textit{let } x_2 = \textit{snd}(x) \textit{ in } M_2 \quad \text{where } x \notin \mathit{fv}(M_2) \\
\textit{let } f(x) = M_1 \textit{ in } M_2 & \stackrel{\text{def}}{=} & \textit{let } f' = \textit{rec}\, f(x).\, M_1 \textit{ in } M_2[f'/f] \quad \text{where } f' \notin \mathit{fv}(M_2)
\end{array}
$$

Figure 2: Sugar for the ReFS syntax

$$\Gamma, x : \sigma \vdash x : \sigma \qquad \Gamma \vdash \mathit{true} : \mathit{bool} \qquad \Gamma \vdash \mathit{false} : \mathit{bool}$$

$$\Gamma \vdash n : \mathit{int} \quad (n \in \mathbb{Z}) \qquad \Gamma \vdash () : \mathit{unit} \qquad \Gamma \vdash \ell : \mathit{loc} \quad (\ell \in \mathit{Loc})$$

$$\frac{\Gamma, f : \sigma \to \sigma', x : \sigma \vdash M : \sigma'}{\Gamma \vdash \mathit{rec}\, f(x).\, M : \sigma \to \sigma'}\,(f, x \notin \Gamma) \qquad \frac{\Gamma \vdash V : \sigma \quad \Gamma \vdash V' : \sigma'}{\Gamma \vdash (V, V') : \sigma \times \sigma'}$$

$$\frac{\Gamma \vdash V : \mathit{bool} \quad \Gamma \vdash M : \sigma \quad \Gamma \vdash M' : \sigma}{\Gamma \vdash \mathit{if}\ V\ \mathit{then}\ M\ \mathit{else}\ M' : \sigma}$$

$$\frac{\Gamma \vdash V : \mathit{int} \quad \Gamma \vdash V' : \mathit{int}}{\Gamma \vdash V\ \mathit{op}\ V' : \gamma}\,(\gamma \in \{\mathit{bool}, \mathit{int}\} \text{ is the result type of } \mathit{op})$$

$$\frac{\Gamma \vdash V : \sigma \times \sigma'}{\Gamma \vdash \mathit{fst}(V) : \sigma} \qquad \frac{\Gamma \vdash V : \sigma \times \sigma'}{\Gamma \vdash \mathit{snd}(V) : \sigma'}$$

$$\frac{\Gamma \vdash V : \mathit{int}}{\Gamma \vdash \mathit{ref}(V) : \mathit{loc}} \qquad \frac{\Gamma \vdash V : \mathit{loc}}{\Gamma \vdash\, !V : \mathit{int}} \qquad \frac{\Gamma \vdash V : \mathit{loc} \quad \Gamma \vdash V' : \mathit{int}}{\Gamma \vdash V := V' : \mathit{unit}}$$

$$\frac{\Gamma \vdash V : \sigma \to \sigma' \quad \Gamma \vdash V' : \sigma}{\Gamma \vdash V V' : \sigma'} \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma, x : \sigma \vdash M' : \sigma'}{\Gamma \vdash \mathit{let}\, x = M\ \mathit{in}\ M' : \sigma'}\,(x \notin \Gamma)$$

Notation. We use the following notation for various collections of well-typed expressions and values.

$$\mathit{Exp}_\sigma(\Gamma) \stackrel{\text{def}}{=} \{M \mid \Gamma \vdash M : \sigma\} \qquad \mathit{Val}_\sigma(\Gamma) \stackrel{\text{def}}{=} \{V \in \mathit{Exp}_\sigma(\Gamma) \mid V \text{ a value}\}$$

$$\mathit{Exp}_\sigma \stackrel{\text{def}}{=} \mathit{Exp}_\sigma(\emptyset) \qquad \mathit{Val}_\sigma \stackrel{\text{def}}{=} \mathit{Val}_\sigma(\emptyset)$$

$$\mathit{Exp} \stackrel{\text{def}}{=} \bigcup\{\mathit{Exp}_\sigma \mid \sigma \text{ a type}\} \qquad \mathit{Val} \stackrel{\text{def}}{=} \bigcup\{\mathit{Val}_\sigma \mid \sigma \text{ a type}\}.$$

Figure 3: ReFS type assignment

avoid the use of environments in the ReFS operational semantics. Variables may be free or bound, while locations are always free. The form $rec\, f(x).\, M$ binds free occurrences of the variables f and x in expression M, and $let\, x = M\, in\, M'$ binds any free occurrences of x in M'. We identify expressions and values up to α-conversion of bound variables. The finite sets of free variables and locations of an expression M are denoted $fv(M)$ and $loc(M)$ respectively. We substitute values for free variables $M[V/x]$ in the usual capture-avoiding way; the restriction to values V arises from the choice of reduced syntax and is appropriate for a call-by-value language.

We only consider expressions that are well-typed. The ReFS types are given in Figure 1: $bool$ and int are the types of booleans and integers respectively; $unit$ is a one-value type; loc is the type of names of integer storage locations, corresponding to the Standard ML type `int ref`; $\sigma \to \sigma'$ and $\sigma \times \sigma$ are function and product types, corresponding to the Standard ML types $\sigma -> \sigma'$ and $\sigma * \sigma'$. For simplicity, we assume that the set Var of variables is partitioned into a family of countably infinite subsets, one for each type: thus each variable x comes with a type σ, and we write $x : \sigma$ to indicate this. The rules for assigning types to expressions are given in Figure 3 and are quite standard. They inductively define a judgement of the form $\Gamma \vdash M : \sigma$, where Γ is a finite subset of Var, M is a ReFS expression, and σ is a ReFS type. The role of Γ in the judgement is to indicate explicitly a set of variables free for substitution in M. Indeed, it is not hard to prove that if $\Gamma \vdash M : \sigma$ is derivable then $fv(M) \subseteq \Gamma$. Most of the time we will be dealing with *closed* expressions, by which we mean expressions with no free variables, but quite possibly involving location constants $\ell \in Loc$.

To further simplify the operational semantics of ReFS we have rolled function abstraction and recursive function declaration into the one form $rec\, f(x).\, M$ which corresponds to the Standard ML value

$$\texttt{fn x} \Rightarrow (\texttt{let fun f x} = \texttt{M in M end}).$$

Figure 2 shows how ordinary lambda abstraction, the call-by-value Y combinator, local recursive function definitions and $while$ loops are all special cases of this construct.

The ReFS operations for manipulating store are exactly as in Standard ML, but restricted to storage of *integer* values. Expression $ref(V)$ allocates local store, placing the integer denoted by V at some fresh location, which is then returned as the value of the expression; operation $!V$ fetches the value stored at the location denoted by V; and $V := V'$ updates it with the integer denoted by V', returning the unit value $()$.

Remark 2.1 (Testing equality of locations). ReFS does not contain a primitive operation $eq : loc \to loc \to bool$ for testing equality of locations (as opposed to equality of their contents). Nevertheless such an operation is definable. For example

$$eq \stackrel{\text{def}}{=} \lambda\, x.\, \lambda\, x'.\, let\, v = !x\, in\, (x := !x' + 1; let\, b = (!x = !x')\, in\, (x := v; b))$$

$$s, V \Downarrow V, s \qquad (\Downarrow\text{val})$$

$$\frac{s, M_b \Downarrow V, s'}{s, \mathit{if}\ b\ \mathit{then}\ M_{\mathit{true}}\ \mathit{else}\ M_{\mathit{false}} \Downarrow V, s'} \qquad (\Downarrow\text{if})$$

$$s, n\ \mathit{op}\ n' \Downarrow c, s \quad \text{if } c = n\ \mathit{op}\ n' \qquad (\Downarrow\text{op})$$

$$s, \mathit{fst}((V, V')) \Downarrow V, s \qquad (\Downarrow\text{fst})$$

$$s, \mathit{snd}((V, V')) \Downarrow V', s \qquad (\Downarrow\text{snd})$$

$$\frac{s, M[\,(\mathit{rec}\ f(x).\ M)/f,\ V/x\,] \Downarrow V', s'}{s, (\mathit{rec}\ f(x).\ M)V \Downarrow V', s'} \qquad (\Downarrow\text{app})$$

$$s, \mathit{ref}(n) \Downarrow \ell, s \otimes (\ell := n) \quad \text{any } \ell \notin \mathit{dom}(s) \qquad (\Downarrow\text{ref})$$

$$s, !\ell \Downarrow n, s \quad \text{if } n = s(\ell) \qquad (\Downarrow\text{get})$$

$$s, \ell := n \Downarrow (), (s; \ell := n) \qquad (\Downarrow\text{set})$$

$$\frac{s, M \Downarrow V, s' \qquad s', M'[V/x] \Downarrow V', s''}{s, \mathit{let}\ x = M\ \mathit{in}\ M' \Downarrow V', s''} \qquad (\Downarrow\text{let})$$

Figure 4: ReFS evaluation rules

has the required evaluation properties with respect to the operational semantics of ReFS introduced below.

Evaluation of expressions

The meaning of ReFS expressions clearly depends on the current contents of memory or *state*. We represent states as finite partial functions from locations to integers $s : \mathit{Loc} \rightharpoonup_{\text{fin}} \mathbb{Z}$, with $\mathit{dom}(s)$ being the locations actually occupied. The empty state is denoted $()$, and for any state s, location ℓ and integer n we write $(s; \ell := n)$ for the updated state defined by

$$\begin{aligned} \mathit{dom}(s; \ell := n) &= \mathit{dom}(s) \cup \{\ell\} \\ (s; \ell := n)(\ell') &= \begin{cases} s(\ell') & \text{if } \ell' \neq \ell, \\ n & \text{if } \ell' = \ell. \end{cases} \end{aligned}$$

In the case that $\ell \notin \mathit{dom}(s)$, we write $s \otimes (\ell := n)$ for $(s; \ell := n)$. More generally, given states s and s' with disjoint domains their *smash product* $s \otimes s'$ is the state

with

$$dom(s \otimes s') \;=\; dom(s) \cup dom(s')$$
$$(s \otimes s')(\ell) \;=\; \begin{cases} s(\ell) & \text{if } \ell \in dom(s), \\ s'(\ell) & \text{if } \ell \in dom(s'). \end{cases}$$

We write Sta for the set of all states; and if $\omega \subseteq Loc$ is a finite set of location constants, we write $Sta(\omega)$ for the subset of Sta consisting of all states s with $dom(s) = \omega$.

Much as in the Definition of Standard ML (Milner, Tofte, and Harper 1990), we give the operational semantics of ReFS via an inductively defined evaluation relation of the form

$$s, M \Downarrow V, s' \tag{2.1}$$

where $s, s' \in Sta$, $M \in Exp$, and $V \in Val$. We consider only *well-formed* judgements, where M and V may be given some common type σ, and all locations used are properly defined: $loc(M) \subseteq dom(s)$ and $loc(V) \subseteq dom(s')$. The rules defining the relation are given in Figure 4 and are all quite standard. We write $s, M \Downarrow$ to indicate termination, i.e. that $s, M \Downarrow V, s'$ holds for some s', V (and hence in particular that $loc(M) \subseteq dom(s)$).

Even taking into account differences in syntax, there are some differences between this operational semantics and the corresponding fragment of the Standard ML definition (Milner, Tofte, and Harper 1990). For one thing, we have eliminated the use of environments in the evaluation relation at the expense of introducing the syntactic operation of substitution. Thus in rule (⇓app), $M[\,(rec\, f(x).\, M)/f,\, V/x\,]$ denotes the result, well-defined up to α-conversion, of simultaneously substituting $rec\, f(x).\, M$ for all free occurrences of f and V for all free occurrences of x, in M. The small price to pay for this approach is the explicit appearance of locations in the syntax of expressions.

More significantly, the reduced syntax has concentrated the sequencing of evaluation in the language down to just one rule: only (⇓let) has more than one hypothesis, and most have none.

Contextual equivalence

We regard two expressions of ReFS as equivalent if they can be used interchangeably in any program without affecting the observable results of program execution. This is formalised by the standard notion of *contextual equivalence*, suitably adapted for the language in hand.

As usual, a *context* $C[-]$ is a ReFS expression in which some subexpressions have been replaced by occurrences of a parameter, or *hole*, '$-$'. The expression resulting from filling the holes with an expression M is denoted by $C[M]$. Since

the holes may occur within the scope of let- and rec-binders, free variables of M may become bound in $C[M]$. This 'capture' of variables means that although the operation of substituting M for '$-$' in $C[-]$ respects α-conversion of M, it does not necessarily respect α-conversion of $C[-]$. Therefore we do not identify contexts up to α-conversion.

In the following definitions of contextual preorder and equivalence, we take convergence at arbitrary type as our basic observable. As it happens, the expressiveness of contexts means that we could have chosen other observations without changing the relations that result: convergence at unit type, or to a specified integer value, would do just as well.

Definition 2.2 (Contextual preorder, contextual equivalence).
Suppose that $M_1, M_2 \in Exp_\sigma(\Gamma)$ are ReFS expressions. We write

$$\Gamma \vdash M_1 \leq M_2 : \sigma \qquad \text{(\textit{contextual preorder})}$$

if for all contexts $C[-]$ such that $C[M_1]$ and $C[M_2]$ are closed terms of the same type it is the case that

$$s, C[M_1] \Downarrow \Rightarrow s, C[M_2] \Downarrow$$

holds for all states s with $loc(C[M_1], C[M_2]) \subseteq dom(s)$. We define

$$\Gamma \vdash M_1 \cong M_2 : \sigma \qquad \text{(\textit{contextual equivalence})}$$

to mean that $\Gamma \vdash M_1 \leq M_2 : \sigma$ and $\Gamma \vdash M_2 \leq M_1 : \sigma$.

It is an easy consequence of these definitions that $\leq$ is reflexive and transitive, and hence that $\cong$ is an equivalence relation; moreover both relations are preserved by all the expression-forming operations of the language (including those that bind free variables).

3 A structurally inductive definition of termination

Before describing the logical relation for ReFS which is the main contribution of this paper, we need to describe the continuation-based termination relation upon which it depends. As mentioned on page 7, the reformulation of termination which we present in this section seems necessary in order to formulate a notion of logical relation that respects both dynamically allocated local state and recursively defined higher-order functions. Apart from this, the structurally inductive definition of termination we give here is very convenient for formalising inductive proofs about contextual equivalence, for the following reason.

Developing properties of ReFS contextual equivalence directly from its definition is not so easy. This is due to the quantification over all possible contexts that occurs in Definition 2.2 together with the nature of the termination relation, $s, M \Downarrow$.

Although it is an inductive (*i.e.* recursively enumerable) subset of $Sta \times Exp$, its definition is not *structurally* inductive. For example, we can derive the rule

$$\frac{s', M'[V/x] \Downarrow}{s, let\ x = M\ in\ M' \Downarrow} \quad \text{if } s, M \Downarrow V, s'$$

but the value expression V used in the substitution $M'[V/x]$ is not a primitive recursive function of the syntax of $let\ x = M\ in\ M'$. As a consequence, the proof methods for contextual equivalence which naturally suggest themselves—induction over the structure of contexts and induction on the derivation of termination from the rules in Figure 4—very often founder for want of a sufficiently strong induction hypothesis. We shall fill this need for stronger induction hypotheses by considering a larger set than $Sta \times Exp$, carving out a subset by structural induction, and exhibiting the termination relation $\{(s, M) \mid s, M \Downarrow\}$ as a retract of this subset. The key ingredient in this strategy is a formal version of continuations.

Continuations

The concept of continuation that we extract from ReFS evaluation is fairly standard. However, our continuations are typed at both argument and result; and we have no need here of continuation *passing*. Continuations take the form of finite lists of expression abstractions $(x)M$, with $\mathcal{I}d$ for the empty list and '$\circ$' for concatenation:

$$K ::= \mathcal{I}d \mid K \circ (x)M\ .$$

Free variables and locations are defined by

$$\begin{aligned} fv(\mathcal{I}d) &= \emptyset & fv(K \circ (x)M) &= fv(K) \cup (fv(M) \setminus \{x\}) \\ loc(\mathcal{I}d) &= \emptyset & loc(K \circ (x)M) &= loc(K) \cup loc(M). \end{aligned}$$

We identify continuations up to α-conversion of bound variables (free occurrences of x in M are bound in $(x)M$).

The *application* $K@M$ of a continuation to an expression is defined by

$$\begin{aligned} \mathcal{I}d@M &\stackrel{\text{def}}{=} M \\ (K \circ (x)M')@M &\stackrel{\text{def}}{=} K@(let\ x = M\ in\ M') \end{aligned}$$

and is an expression (well-defined up to α-conversion). This notion of application gives a tie-up with *evaluation contexts*. For ReFS, with its reduced form of syntax, these are simply the subset of expression contexts given by

$$E[-] ::= [-] \mid let\ x = E[-]\ in\ M.$$

For any such $E[-]$, there is a continuation K such that $E[M] \equiv K@M$ for all expressions M, and conversely every continuation has a matching evaluation

context. Evaluation contexts were originally derived (as 'unlabelled sk-contexts') from continuations in (Felleisen and Friedman 1986).

To each continuation we assign a type $\sigma \circ\!\!\to \sigma'$, where the notation is meant to suggest the fact that evaluation contexts would give rise to *strict* (continuous) functions in a denotational semantics. The rules for types are as follows:

$$\Gamma \vdash \mathcal{I}d : \sigma \circ\!\!\to \sigma \qquad \frac{\Gamma \vdash K : \sigma_2 \circ\!\!\to \sigma_3}{\Gamma \vdash K \circ (x)M : \sigma_1 \circ\!\!\to \sigma_3} \text{ if } \Gamma, x : \sigma_1 \vdash M : \sigma_2$$

Note that if $\Gamma \vdash K : \sigma \circ\!\!\to \sigma'$ and $\Gamma \vdash M : \sigma$, then $\Gamma \vdash K@M : \sigma'$. We collect typed continuations into a range of indexed sets.

$$Cont_{\sigma,\sigma'}(\Gamma) \stackrel{\text{def}}{=} \{K \mid \Gamma \vdash K : \sigma \circ\!\!\to \sigma'\} \quad Cont_\sigma(\Gamma) \stackrel{\text{def}}{=} \bigcup\{Cont_{\sigma,\sigma'}(\Gamma) \mid \sigma' \text{ a type}\}$$
$$Cont_\sigma \stackrel{\text{def}}{=} Cont_\sigma(\emptyset) \qquad Cont \stackrel{\text{def}}{=} \bigcup\{Cont_\sigma \mid \sigma \text{ a type}\}.$$

Termination

We are now ready to give our *structurally* defined termination relation. This will be an inductively defined subset of $Sta \times Cont \times Exp$ and we write

$$\langle s, K, M\rangle\!\downarrow$$

to indicate that (s, K, M) is in the subset. As usual we consider only well-formed judgements, here requiring that $M \in Exp_\sigma$ and $K \in Cont_\sigma$ for some type σ, and that $loc(K, M) \subseteq dom(s)$. Figure 5 gives the rules defining the relation. Notice that these are now properly structurally inductive, with a simple syntactic connection between the conclusion and hypothesis of each rule.

Theorem 3.1 (Termination). *The two termination relations correspond in the sense that*

$$\langle s, K, M\rangle\!\downarrow \;\Leftrightarrow\; s, K@M \Downarrow . \tag{3.1}$$

In particular, one has

$$\langle s, \mathcal{I}d, M\rangle\!\downarrow \;\Leftrightarrow\; s, M \Downarrow \tag{3.2}$$
$$\langle s, K, M\rangle\!\downarrow \;\Leftrightarrow\; \exists V, s'\,(s, M \Downarrow V, s' \;\&\; \langle s', K, V\rangle\!\downarrow). \tag{3.3}$$

Proof. One way to prove these properties is to note that

$$\langle s, K, M\rangle\!\downarrow \Leftrightarrow \exists s', V\,(\langle s, K, M\rangle \to^* \langle s', \mathcal{I}d, V\rangle).$$

where $\to^*$ is the reflexive-transitive closure of a suitable transition relation $\to$ between configurations. One can then establish (3.1)–(3.3) via a series of inductions involving $\Downarrow$ and $\to$. We omit the details. □

$$\langle s, \mathcal{I}d, V\rangle\downarrow \qquad (\downarrow\text{val}_1)$$

$$\frac{\langle s, K, M[V/x]\rangle\downarrow}{\langle s, K \circ (x)M, V\rangle\downarrow} \qquad (\downarrow\text{val}_2)$$

$$\frac{\langle s, K, M_b\rangle\downarrow}{\langle s, K, \mathit{if}\ b\ \mathit{then}\ M_{\mathit{true}}\ \mathit{else}\ M_{\mathit{false}}\rangle\downarrow} \qquad (\downarrow\text{if})$$

$$\frac{\langle s, K, c\rangle\downarrow}{\langle s, K, n\ \mathit{op}\ n'\rangle\downarrow} \quad \text{if } c = n\ \mathit{op}\ n' \qquad (\downarrow\text{op})$$

$$\frac{\langle s, K, V\rangle\downarrow}{\langle s, K, \mathit{fst}((V, V'))\rangle\downarrow} \qquad (\downarrow\text{fst})$$

$$\frac{\langle s, K, V'\rangle\downarrow}{\langle s, K, \mathit{snd}((V, V'))\rangle\downarrow} \qquad (\downarrow\text{snd})$$

$$\frac{\langle s, K, M[\,(\mathit{rec}\, f(x).\, M)/f,\, V/x\,]\rangle\downarrow}{\langle s, K, (\mathit{rec}\, f(x).\, M)V\rangle\downarrow} \qquad (\downarrow\text{app})$$

$$\frac{\langle s \otimes (\ell := n), K, \ell\rangle\downarrow}{\langle s, K, \mathit{ref}(n)\rangle\downarrow} \quad \text{if } \ell \notin \mathit{dom}(s) \cup \mathit{loc}(K) \qquad (\downarrow\text{ref})$$

$$\frac{\langle s, K, n\rangle\downarrow}{\langle s, K, !\ell\rangle\downarrow} \quad \text{if } n = s(\ell) \qquad (\downarrow\text{get})$$

$$\frac{\langle s; \ell := n, K, ()\rangle\downarrow}{\langle s, K, \ell := n\rangle\downarrow} \qquad (\downarrow\text{set})$$

$$\frac{\langle s, K \circ (x)M', M\rangle\downarrow}{\langle s, K, \mathit{let}\, x = M\ \mathit{in}\ M'\rangle\downarrow} \qquad (\downarrow\text{let})$$

Figure 5: Continuation-based termination relation

The unwinding theorem

In contrast to $\Downarrow$, the *structural* nature of the termination relation $\downarrow$ enables many properties of the contextual preorder and equivalence relations to be proved in a rather straightforward way, via an induction on the derivation of $\langle s, K, M\rangle\!\downarrow$. As an illustration, we give such a proof for the 'unwinding theorem' for recursive function values in ReFS, which provides a syntactic analogue of Dana Scott's induction principle for least fixed points and which is needed in the proof of the Fundamental Property of the logical relation introduced in the next section. Such theorems have been proved by several different people in various contexts: see for example (Mason, Smith, and Talcott 1996).

We fix some closed recursively defined function value $rec\, f(x).\, F \in Val_{\sigma\to\sigma'}$ and define the following abbreviations:

$$\begin{aligned} \Omega &\stackrel{\text{def}}{=} (rec\, f(x).\, fx)() \\ F_0 &\stackrel{\text{def}}{=} \lambda\, x.\, \Omega \\ F_{n+1} &\stackrel{\text{def}}{=} \lambda\, x.\, F[F_n/f] \\ F_\omega &\stackrel{\text{def}}{=} rec\, f(x).\, F\,. \end{aligned}$$

Each F_n is a finite unwinding of the full function F_ω. The essence of the following theorem is that these finite approximations provide all the observable behaviour of F_ω itself.

Theorem 3.2 (Unwinding). *For any $M \in Exp_{\sigma''}(f{:}\sigma\to\sigma')$ we have*

$$s, M[F_\omega/f]\Downarrow \Leftrightarrow \exists n \in \mathbb{N}\,(s, M[F_n/f]\Downarrow).$$

Equivalently, for any $K \in Cont_{\sigma\to\sigma'}$

$$\langle s, K, F_\omega\rangle\!\downarrow \Leftrightarrow \exists n \in \mathbb{N}\,\langle s, K, F_n\rangle\!\downarrow.$$

Proof. The two statements in the theorem are equivalent by Theorem 3.1, noting that for any K and F, $K@F$ is of the form $M[F/f]$ for some M, and conversely that when F is a value then $s, M[F/f]\Downarrow$ holds if and only if $\langle s, \mathcal{I}d \circ (x)M, F\rangle\!\downarrow$. The theorem (in its second formulation) follows from parts (iii) and (iv) of the following lemma, by taking $K' = K$ and $M' = g$. □

Lemma 3.3. *For all $M' \in Exp_{\sigma_1}(g{:}\sigma\to\sigma')$, $K' \in Cont_{\sigma_1}(g{:}\sigma\to\sigma')$, and $s \in Sta$, if $loc(M', K', F) \subseteq dom(s)$ then*

(i) *for all $G \in Val_{\sigma\to\sigma'}$ with $loc(G) \subseteq dom(s)$*

$$\langle s, K'[F_0/g], M'[F_0/g]\rangle\!\downarrow \;\Rightarrow\; \langle s, K'[G/g], M'[G/g]\rangle\!\downarrow$$

(ii) *for all $n \in \mathbb{N}$*

$$\langle s, K'[F_n/g], M'[F_n/g]\rangle\!\downarrow \;\Rightarrow\; \langle s, K'[F_{n+1}/g], M'[F_{n+1}/g]\rangle\!\downarrow$$

(iii) $\langle s, K'[F_\omega/g], M'[F_\omega/g]\rangle\downarrow \;\Rightarrow\; \exists n \in \mathbb{N}.\langle s, K'[F_n/g], M'[F_n/g]\rangle\downarrow$

(iv) *for all* $n \in \mathbb{N}$

$$\langle s, K'[F_n/g], M'[F_n/g]\rangle\downarrow \;\Rightarrow\; \langle s, K'[F_\omega/g], M'[F_\omega/g]\rangle\downarrow.$$

Proof. (i) is proved by induction on the derivation of $\langle s, K'[F_0/g], M'[F_0/g]\rangle\downarrow$ from the rules in Figure 5. More precisely, one shows that the set of machine states

$$\{\, \langle s, K, M\rangle \mid \forall K', M'\,(K = K'[F_0/g] \;\&\; M = M'[F_0/g] \\ \Rightarrow \forall G\, \langle s, K'[G/g], M'[G/g]\rangle\downarrow)\,\}$$

is closed under those rules. The only non-straightforward case is ($\downarrow$app), where one uses the easily verified fact that $\langle s, K, \Omega\rangle\downarrow$ cannot hold for any s and K. We omit the details.

(ii) is proved by induction on n, with part (i) providing the base case of $n = 0$.

For (iii), again one works by induction on the proof of termination, showing that the set of machine states

$$\mathbb{T} = \{\, \langle s, K, M\rangle \mid \forall K', M'\,(K = K'[F_\omega/g] \;\&\; M = M'[F_\omega/g] \\ \Rightarrow \exists n \in \mathbb{N}\, \langle s, K'[F_n/g], M'[F_n/g]\rangle\downarrow)\,\}$$

is closed under the rules of Figure 5 generating $\downarrow$. As for part (i), the only difficult case is closure under the application rule ($\downarrow$app). For that, suppose we have

$$\langle s, K, M[\,(\mathit{rec}\, f(x).\, M)/f,\, V/x\,]\rangle \in \mathbb{T}. \tag{3.4}$$

Then we have to show that $\langle s, K, (\mathit{rec}\, f(x).\, M)V\rangle \in \mathbb{T}$, i.e. that if

$$K = K'[F_\omega/g] \quad \text{and} \quad (\mathit{rec}\, f(x).\, M)V = M'[F_\omega/g] \tag{3.5}$$

then $\langle s, K'[F_n/g], M'[F_n/g]\rangle\downarrow$ holds for some finite n. Now (3.5) must hold because $M' = V_1V_2$ for some values V_1 and V_2 such that

$$\mathit{rec}\, f(x).\, M = V_1[F_\omega/g] \quad \text{and} \quad V = V_2[F_\omega/g].$$

The first of these can occur in two situations:

(a) $V_1 = g$ and $\mathit{rec}\, f(x).\, M = F_\omega$, thus $M = F$.

(b) $V_1 = \mathit{rec}\, f(x).\, M_1$ for some M_1 with $M = M_1[F_\omega/g]$.

The proof in case (b) is straightforward and we omit it. In case (a) we now have $M[\,(\mathit{rec}\, f(x).\, M)/f,\, V/x\,] = F[\,g/f,\, V_2/x\,][F_\omega/g]$ and so by (3.4) there is some finite m with $\langle s, K'[F_m/g], F[\,g/f,\, V_2/x\,][F_m/g]\rangle\downarrow$. Using the definition of F_{m+1}

and rule (↓app) gives $\langle s, K'[F_m/g], F_{m+1}(V_2[F_m/g])\rangle\!\downarrow$ and hence by part (ii) also $\langle s, K'[F_{m+1}/g], F_{m+1}(V_2[F_{m+1}/g])\rangle\!\downarrow$. But

$$F_{m+1}(V_2[F_{m+1}/g]) = (V_1V_2)[F_{m+1}/g] = M'[F_{m+1}/g]$$

and so we have the desired conclusion that $\langle s, K'[F_n/g], M'[F_n/g]\rangle\!\downarrow$ holds for some n (namely $n = m + 1$).

The closure of $\mathbb{T}$ under the other rules of Figure 5 requires the same straightforward reasoning as for case (b), and we omit the details.

Finally, part (iv) of the lemma is once again proved by an induction over $\downarrow$: one shows that the set of machine states

$$\{\, \langle s, K, M\rangle \mid \forall K', M', n\, (K = K'[F_n/g] \;\&\; M = M'[F_n/g] \\ \Rightarrow \langle s, K'[F_\omega/g], M'[F_\omega/g]\rangle\!\downarrow)\,\}$$

is closed under the rules generating $\downarrow$. Again, the only non-routine case is for (↓app) and for that the proof is very much as for part (iii), with two distinct cases (a) and (b). We omit the details. □

Note that the imperative features of ReFS have little rôle in the proof of unwinding. If we had storage of non-ground data, these features would play a greater part, but the same proof methods would still work.

4 A parametric logical relation

In this section we define a family of binary relations between ReFS expressions (of equal type) parameterised by relations between states and establish its relationship to contextual equivalence. We prove a 'Fundamental Property' typical of logical relations (Theorem 4.9). This is the main technical result of the paper and it draws heavily upon the work of the previous section. From the Fundamental Property we easily deduce an extensionality result for ReFS contextual equivalence (Theorem 4.10) that includes the 'ciu' theorem of Mason and Talcott (1992b). So we get proofs for a range of basic contextual equivalences that are the usual consequences of the 'ciu' theorem. However, our extensionality theorem also characterises contextual equivalence in terms of the logical relation (with the state-relation parameter instantiated to the identity). In the next section we shall show that this characterisation can be used to give quite straightforward proofs for some examples of contextual equivalence which are not easily seen to be direct consequences of the 'ciu' theorem.

Definitions

We begin by defining a variety of relations between elements of our ReFS language, starting with states.

Definition 4.1 (State relations). Given finite subsets $\omega_1, \omega_2 \subseteq Loc$, a *state relation* from ω_1 to ω_2 is a non-empty subset $r \subseteq Sta(\omega_1) \times Sta(\omega_2)$. (Recall from page 13 that $Sta(\omega)$ denotes the set of all states with domain of definition equal to ω.) We write

$$Rel(\omega_1, \omega_2)$$

for the set of all such relations. Given $r \in Rel(\omega_1, \omega_2)$, we refer to ω_1 and ω_2 as the domain and codomain of r respectively. (Note that since we are assuming that any state relation r is in particular non-empty,[3] its domain and codomain are uniquely determined.)

For any finite subset $\omega \subseteq Loc$, the *identity state relation* on ω is

$$id_\omega \stackrel{\text{def}}{=} \{(s, s) \mid dom(s) = \omega\}.$$

Given two state relations relations $r \in Rel(\omega_1, \omega_2)$ and $r' \in Rel(\omega_1', \omega_2')$ with $\omega_i \cap \omega_i' = \emptyset$ ($i = 1, 2$), their *smash product* $r \otimes r' \in Rel(w_1 \cup \omega_1', \omega_2 \cup \omega_2')$ is defined using the smash product of states defined in Section 2:

$$r \otimes r' \stackrel{\text{def}}{=} \{(s_1 \otimes s_1', s_2 \otimes s_2') \mid (s_1, s_2) \in r \;\&\; (s_1', s_2') \in r'\}.$$

It is straightforward to show that

$$id_\omega \otimes id_{\omega'} = id_{\omega \cup \omega'} \quad (w \cap w' = \emptyset) \qquad\qquad r \otimes r' = r' \otimes r$$
$$id_\emptyset \otimes r = r \qquad\qquad r \otimes (r' \otimes r'') = (r \otimes r') \otimes r''$$

with the last three in particular following from the corresponding property of $\otimes$ on states.

We say that a state relation r' *extends* another one r, and write $r' \rhd r$, if $r' = r \otimes r''$ for some r''. It follows from the above properties of the smash product $\otimes$ that the extension relation $\rhd$ is a partial order.

Suppose we have two configurations $\langle s, K, M\rangle$ and $\langle s', K', M'\rangle$ of the abstract machine described in Section 3. We say that they are *convergence equivalent*, written

$$\langle s, K, M\rangle \updownarrow \langle s', K', M'\rangle$$

if they are both well formed, *i.e.* $loc(KM) \subseteq dom(s)$ and $loc(K'M') \subseteq dom(s')$, and they converge or diverge together:

$$\langle s, K, M\rangle{\downarrow} \Leftrightarrow \langle s', K', M'\rangle{\downarrow}.$$

[3] This is merely a technical convenience which, amongst other things, simplifies the definition of the logical relation at ground types.

Definition 4.2 (A parametric logical relation for ReFS).
For each state relation $r \in Rel(\omega_1, \omega_2)$ and each type σ we define three binary relations:

$$\begin{aligned}\mathcal{E}_\sigma(r) &\subseteq Exp_\sigma(\omega_1) \times Exp_\sigma(\omega_2)\\ \mathcal{K}_\sigma(r) &\subseteq Cont_\sigma(\omega_1) \times Cont_\sigma(\omega_2)\\ \mathcal{V}_\sigma(r) &\subseteq Val_\sigma(\omega_1) \times Val_\sigma(\omega_2).\end{aligned}$$

Here $Exp_\sigma(\omega)$ denotes the set of closed ReFS expressions of type σ involving location constants in the finite set ω; similarly for continuations $Cont_\sigma(\omega)$ and values $Val_\sigma(\omega)$. We make the definitions of these relations for all r simultaneously. The first relation, between expressions, is defined in terms of the second:

$$(M_1, M_2) \in \mathcal{E}_\sigma(r) \stackrel{\text{def}}{\Leftrightarrow} \forall r' \rhd r, (s_1, s_2) \in r', (K_1, K_2) \in \mathcal{K}_\sigma(r')\,. \\ \langle s_1, K_1, M_1\rangle \updownarrow \langle s_2, K_2, M_2\rangle. \quad (4.1)$$

The second relation, on continuations, is defined in terms of the third:

$$(K_1, K_2) \in \mathcal{K}_\sigma(r) \stackrel{\text{def}}{\Leftrightarrow} \forall r' \rhd r, (s_1, s_2) \in r', (V_1, V_2) \in \mathcal{V}_\sigma(r')\,. \\ \langle s_1, K_1, V_1\rangle \updownarrow \langle s_2, K_2, V_2\rangle. \quad (4.2)$$

The final relation, between values, is defined by induction on the structure of the type σ:

$$(c_1, c_2) \in \mathcal{V}_\sigma(r) \stackrel{\text{def}}{\Leftrightarrow} c_1 = c_2 \quad \text{for } \sigma \in \{unit, bool, int\} \quad (4.3)$$

$$(\ell_1, \ell_2) \in \mathcal{V}_{loc}(r) \stackrel{\text{def}}{\Leftrightarrow} (!\ell_1, !\ell_2) \in \mathcal{E}_{int}(r) \\ \;\&\; \forall n \in \mathbb{Z}\,.\,(\ell_1 := n, \ell_2 := n) \in \mathcal{E}_{unit}(r) \quad (4.4)$$

$$(V_1, V_2) \in \mathcal{V}_{\sigma\times\sigma'}(r) \stackrel{\text{def}}{\Leftrightarrow} (fst(V_1), fst(V_2)) \in \mathcal{E}_\sigma(r) \\ \;\&\; (snd(V_1), snd(V_2)) \in \mathcal{E}_{\sigma'}(r) \quad (4.5)$$

$$(V_1, V_2) \in \mathcal{V}_{\sigma\to\sigma'}(r) \stackrel{\text{def}}{\Leftrightarrow} \forall r' \rhd r, (W_1, W_2) \in \mathcal{V}_\sigma(r')\,. \\ (V_1W_1, V_2W_2) \in \mathcal{E}_{\sigma'}(r'). \quad (4.6)$$

We call this family of relations 'logical' simply because it relates function values if, roughly speaking, they map related arguments to related results. This is the characteristic feature of a wide range of relations used in connection with the lambda calculus which ever since (Plotkin 1973, 1980) have been called 'logical relations'.

Note. It is possible to simplify Definition 4.2 by replacing the use of an arbitrary extension $r' \rhd r$ by r itself in the defining clauses for $\mathcal{E}_\sigma(r)$ and $\mathcal{K}_\sigma(r)$ (but not $\mathcal{V}_{\sigma\to\sigma'}(r)$). This simplification depends partly upon the 'flat' nature of state in ReFS and since we have an eye to generalisations of ReFS (see Section 6), we chose not to build it into the definition. Furthermore, this simplification would complicate the proof of the following property.

Lemma 4.3 (Weakening). *Extending a state relation preserves existing relations between expressions, continuations and values: if $r' \rhd r$ then*

$$(M_1, M_2) \in \mathcal{E}_\sigma(r) \Rightarrow (M_1, M_2) \in \mathcal{E}_\sigma(r')$$
$$(K_1, K_2) \in \mathcal{K}_\sigma(r) \Rightarrow (K_1, K_2) \in \mathcal{K}_\sigma(r')$$
$$(V_1, V_2) \in \mathcal{V}_\sigma(r) \Rightarrow (V_1, V_2) \in \mathcal{V}_\sigma(r')$$

Proof. Clauses (4.1) and (4.2) of Definition 4.2 specify $\mathcal{E}_\sigma(r)$ and $\mathcal{K}_\sigma(r)$ by quantifying over all extensions $r' \rhd r$ and so the first two parts are immediate (since $\rhd$ is a preorder). The third part concerning values then follows from the first, matching the way clauses (4.3) to (4.6) define $\mathcal{V}_\sigma(r)$ in terms of $\mathcal{E}_\sigma(r)$. □

The definition of the relations $\mathcal{E}_\sigma(r)$ on expressions and $\mathcal{V}_\sigma(r)$ on values are quite different; nevertheless they agree in that values can be considered as expressions without changing their relations, as the following lemma shows.

Lemma 4.4 (Coincidence). *Relations $\mathcal{E}_\sigma(r)$ and $\mathcal{V}_\sigma(r)$ coincide on values: for $V_1, V_2 \in Val_\sigma$*

$$(V_1, V_2) \in \mathcal{E}_\sigma(r) \Leftrightarrow (V_1, V_2) \in \mathcal{V}_\sigma(r).$$

Proof. The direction from right to left follows at once from the definition of $\mathcal{E}_\sigma(r)$ and $\mathcal{K}_\sigma(r)$ given in clauses (4.1) and (4.2), together with Lemma 4.3. From left to right, we proceed by cases on the structure of type σ.

Case $\sigma = unit$ is trivial, as the only unit value is $()$ and $((), ()) \in \mathcal{V}_{unit}(r)$ for any r.

Case $\sigma = bool$**.** Consider the continuation

$$K = \mathcal{I}d \circ (x)(\mathit{if}\ x\ \mathit{then}\ ()\ \mathit{else}\ \Omega)$$

where Ω is the non-terminating expression used in the Unwinding Theorem 3.2. From the definition of $\mathcal{V}_{bool}$ and $\mathcal{K}_{bool}$ it is not hard to show that $(K, K) \in \mathcal{K}_{bool}(r)$. Suppose then that $b_1, b_2 \in \{true, false\}$ with $(b_1, b_2) \in \mathcal{E}_{bool}(r)$. Since state relations are by definition non-empty, we can choose some $(s_1, s_2) \in r$. Then we have $\langle s_1, K, b_1\rangle \updownarrow \langle s_2, K, b_2\rangle$ and hence

$$\langle s_1, \mathcal{I}d, \mathit{if}\ b_1\ \mathit{then}\ ()\ \mathit{else}\ \Omega\rangle \updownarrow \langle s_2, \mathcal{I}d, \mathit{if}\ b_2\ \mathit{then}\ ()\ \mathit{else}\ \Omega\rangle.$$

Since $\langle s_i, \mathcal{I}d, \Omega\rangle\not\downarrow$ and $\langle s_i, \mathcal{I}d, ()\rangle\downarrow$ it follows that $b_1 = b_2$, and thus $(b_1, b_2) \in \mathcal{V}_{bool}(r)$ as required.

Case $\sigma = int$ is similar to the previous case, using the continuation

$$K = \mathcal{I}d \circ (x)(\mathit{let}\ y = (V_1 = x)\ \mathit{in}\ \mathit{if}\ y\ \mathit{then}\ ()\ \mathit{else}\ \Omega)$$

where V_1 is one of the integer values involved.

Case $\sigma = loc$**.** Suppose that $(\ell_1, \ell_2) \in \mathcal{E}_{loc}(r)$. We need to prove that $(!\ell_1, !\ell_2) \in \mathcal{E}_{int}(r)$ and also $((\ell_1 := n), (\ell_2 := n)) \in \mathcal{E}_{unit}(r)$ for every $n \in \mathbb{Z}$. Suppose that at some extension $r' \rhd r$ we have $(s_1, s_2) \in r'$ and $(K_1, K_2) \in \mathcal{K}_{int}(r')$. For the first property, we have to show that $\langle s_1, K_1, !\ell_1\rangle \updownarrow \langle s_2, K_2, !\ell_2\rangle$. But it is not hard to verify that $((K_1 \circ (x)!x), (K_2 \circ (x)!x)) \in \mathcal{K}_{loc}(r')$, and then because $(\ell_1, \ell_2) \in \mathcal{E}_{loc}(r)$ we obtain

$$\langle s_1, (K_1 \circ (x)!x), \ell_1\rangle \updownarrow \langle s_2, (K_2 \circ (x)!x), \ell_2\rangle$$

and hence also $\langle s_1, K_1, !\ell_1\rangle \updownarrow \langle s_2, K_2, !\ell_2\rangle$, as required. The argument that $(\ell_1 := n, \ell_2 := n) \in \mathcal{E}_{unit}(r)$ is similar.

Case $\sigma = \sigma_1 \times \sigma_2$**.** The proof is as in the previous case, this time taking $(K_1, K_2) \in \mathcal{K}_{\sigma_1}(r')$ to $(K_1 \circ (x)\, fst(x), K_2 \circ (x)\, fst(x)) \in \mathcal{K}_{\sigma}(r')$ and similarly with snd.

Case $\sigma = \sigma_1 \to \sigma_2$**.** The proof takes the same form again, based on the observation that if $r'' \rhd r' \rhd r$ and $(K_1, K_2) \in \mathcal{K}_{\sigma_2}(r'')$ then

$$(\,(K_1 \circ (x)xW_1),\ (K_2 \circ (x)xW_2)\,) \in \mathcal{K}_{\sigma_1 \to \sigma_2}(r'')$$

for all $(W_1, W_2) \in \mathcal{V}_{\sigma_1}(r')$.

These last three cases all use the weakening results of Lemma 4.3. □

Definition 4.5 (Extension of the logical relation to open expressions).
Given two open expressions $M_1, M_2 \in Exp_\sigma(\Gamma)$ with $loc(M_i) \subseteq \omega_i$ $(i = 1, 2)$ and a state relation $r \in Rel(\omega_1, \omega_2)$ we write

$$\Gamma \vdash M_1 \,\{r\}\, M_2 : \sigma$$

to mean that for all extensions $r' \rhd r$ and values $\{V_{1x}, V_{2x} \in Val_{\sigma'} \mid x : \sigma' \in \Gamma\}$ we have

$$(\forall x : \sigma' \in \Gamma\,.\,(V_{1x}, V_{2x}) \in \mathcal{V}_{\sigma'}(r')) \Rightarrow (M_1[\vec{V_1}/\vec{x}], M_2[\vec{V_2}/\vec{x}]) \in \mathcal{E}_\sigma(r').$$

In particular $\emptyset \vdash M_1 \,\{r\}\, M_2 : \sigma$ holds if and only if $(M_1, M_2) \in \mathcal{E}_\sigma(r)$, thanks to Lemma 4.3: so this is indeed an extension of the original logical relation.

The following structural properties of this relation are automatic from the above definition.

$$\Gamma \vdash M_1 \,\{r\}\, M_2 : \sigma \Rightarrow \Gamma\Gamma' \vdash M_1 \,\{r\}\, M_2 : \sigma$$
$$\Gamma \vdash M_1 \,\{r\}\, M_2 : \sigma \Rightarrow \Gamma \vdash M_1 \,\{r \otimes r'\}\, M_2 : \sigma$$

$$(\Gamma \vdash V_1 \,\{r\}\, V_2 : \sigma)\ \&\ (\Gamma, x : \sigma \vdash M_1 \,\{r\}\, M_2 : \sigma') \Rightarrow$$
$$\Gamma \vdash M_1[V_1/x] \,\{r\}\, M_2[V_2/x] : \sigma' \qquad (V_1, V_2 \in Val_\sigma(\Gamma)).$$

In addition to the logical relation, we shall use the following equivalence between expressions.

Definition 4.6 ('ciu' equivalence). Given $M_1, M_2 \in Exp_\sigma$ we write

$$M_1 \cong^{ciu} M_2 : \sigma$$

to mean that for all $s \in Sta$ and $K \in Cont_\sigma$ with $loc(K, M_1, M_2) \subseteq dom(s)$,

$$\langle s, K, M_1\rangle \updownarrow \langle s, K, M_2\rangle.$$

We extend this to open expressions by value substitution: for $M_1, M_2 \in Exp_\sigma(\Gamma)$,

$$\Gamma \vdash M_1 \cong^{ciu} M_2 : \sigma$$

means that for all $\{V_x \in Val_{\sigma'} \mid x : \sigma' \in \Gamma\}$ we have $M_1[\vec{V}/\vec{x}] \cong^{ciu} M_2[\vec{V}/\vec{x}] : \sigma$. (Note that the V_x may use locations not mentioned in M_1 or M_2.)

This relation '$\cong^{ciu}$' coincides with the ciu-equivalence of (Mason and Talcott 1991a; Talcott 1997). This is because the results of Section 3 imply that $\Gamma \vdash M_1 \cong^{ciu} M_2 : \sigma$ holds if and only if for all suitable value substitutions $\{V_x \mid x \in \Gamma\}$, evaluation contexts $E[-]$ and states s:

$$s, E[M_1[\vec{V}/\vec{x}]] \Downarrow \;\Leftrightarrow\; s, E[M_2[\vec{V}/\vec{x}]] \Downarrow .$$

These are the appropriate 'closed instantiations of uses' for ReFS expressions. The next result shows that the logical relations respect ciu-equivalence.

Lemma 4.7 (Composition). *The logical relation is closed under composition with ciu-equivalence. For closed terms we have*

$$\big(M_1' \cong^{ciu} M_1 \;\&\; (M_1, M_2) \in \mathcal{E}_\sigma(r) \;\&\; M_2 \cong^{ciu} M_2'\big) \;\Rightarrow\; (M_1', M_2') \in \mathcal{E}_\sigma(r)$$

and more generally for open ones

$$\big(\Gamma \vdash M_1' \cong^{ciu} M_1 \;\&\; \Gamma \vdash M_1 \,\{r\}\, M_2 \;\&\; \Gamma \vdash M_2 \cong^{ciu} M_2'\big) \;\Rightarrow\; \Gamma \vdash M_1' \,\{r\}\, M_2'.$$

Proof. For the first property, suppose that $r' \rhd r$ and that we have states $(s_1, s_2) \in r'$ and continuations $(K_1, K_2) \in \mathcal{K}_\sigma(r')$. Then

$$\langle s_1, K_1, M_1'\rangle \updownarrow \langle s_1, K_1, M_1\rangle \updownarrow \langle s_2, K_2, M_2\rangle \updownarrow \langle s_2, K_2, M_2'\rangle$$

and the result follows by transitivity of '$\updownarrow$'. For open expressions, we consider values $\{V_{1x}, V_{2x} \in Val_{\sigma'} \mid x : \sigma' \in \Gamma\}$ with each $(V_{1x}, V_{2x}) \in \mathcal{V}_{\sigma'}(r')$ and deduce that

$$M_1'[\vec{V}_1/\vec{x}] \cong^{ciu} M_1[\vec{V}_1/\vec{x}] \;\&\; M_2[\vec{V}_2/\vec{x}] \cong^{ciu} M_2'[\vec{V}_2/\vec{x}] \;\&\; (M_1[\vec{V}_1/\vec{x}], M_2[\vec{V}_2/\vec{x}]) \in \mathcal{E}_\sigma(r').$$

The first part now gives $(M_1'[\vec{V}_1/\vec{x}], M_2'[\vec{V}_2/\vec{x}]) \in \mathcal{E}_\sigma(r')$ and so $\Gamma \vdash M_1' \,\{r\}\, M_2'$ as desired. □

Fundamental property of the relation

We aim to prove the 'Fundamental Property' of the logical relation introduced in Definition 4.2. Roughly speaking, this says that the relations $Exp_\sigma(r)$ are preserved by the various operations in the ReFS language; the precise statement is given below in Theorem 4.9.

The reader familiar with previous work on relational parametricity for languages with storage locations may be surprised that such a property holds without some restriction on the parameterising relations r. O'Hearn and Tennent (1993, Section 6) sketch a construction for a language like ReFS in which the parameter is a binary relation between states equipped with a partial bijection between the underlying sets of locations, which together must satisfy some simple conditions to do with assignment and look-up. The reason given for the use of the extra information of a partial bijection is to ensure that the operation for location-equality testing preserves the parametric logical relation. But in fact use of partial bijections is superfluous. We will establish the preservation property for all expressions in ReFS with respect to the logical relation simply parameterised by state relations; and we noted in Remark 2.1 that location-equality testing is definable in ReFS—so in particular it preserves the logical relation. In this respect ReFS is simpler than the 'nu-calculus' studied in (Pitts and Stark 1993) where location-equality testing is not definable in terms of the rest of the language and use of partial bijections between locations is unavoidable.

Proposition 4.8. *The logical relation is preserved by all the expression-forming operations of the ReFS language.*

(i) $\Gamma, x : \sigma \vdash x \,\{r\}\, x : \sigma$.

(ii) $\Gamma \vdash () \,\{r\}\, () : unit$.

(iii) $\Gamma \vdash b \,\{r\}\, b : bool$, *for each* $b \in \{true, false\}$.

(iv) $\Gamma \vdash n \,\{r\}\, n : int$, *for any* $n \in \mathbb{Z}$.

(v) $\Gamma \vdash \ell \,\{r \otimes id_{\{\ell\}}\}\, \ell : loc$, *where* $\ell \notin dom(r) \cup cod(r)$.

(vi) *If* $\Gamma \vdash V_1 \,\{r\}\, V_2 : \sigma$ *and* $\Gamma \vdash V_1' \,\{r\}\, V_2' : \sigma'$ *then* $\Gamma \vdash (V_1, V_1') \,\{r\}\, (V_2, V_2') : \sigma \times \sigma'$.

(vii) *If* $\Gamma \vdash V_1 \,\{r\}\, V_2 : bool$, $\Gamma \vdash M_1 \,\{r\}\, M_2 : \sigma$ *and* $\Gamma \vdash M_1' \,\{r\}\, M_2' : \sigma$ *then* $\Gamma \vdash (if\ V_1\ then\ M_1\ else\ M_1') \,\{r\}\, (if\ V_2\ then\ M_2\ else\ M_2') : \sigma$.

(viii) *If* $\Gamma \vdash V_1 \,\{r\}\, V_2 : int$ *and* $\Gamma \vdash V_1' \,\{r\}\, V_2' : int$ *then* $\Gamma \vdash (V_1\ op\ V_1') \,\{r\}\, (V_2\ op\ V_2') : \gamma$, *where* γ *is the result type of* op.

(ix) *If* $\Gamma \vdash V_1 \,\{r\}\, V_2 : \sigma \times \sigma'$ *then* $\Gamma \vdash fst(V_1) \,\{r\}\, fst(V_2) : \sigma$ *and* $\Gamma \vdash snd(V_1) \,\{r\}\, snd(V_2) : \sigma'$.

(x) *If* $\Gamma \vdash V_1 \ \{r\} \ V_2 : \mathit{int}$ *then* $\Gamma \vdash \mathit{ref}(V_1) \ \{r\} \ \mathit{ref}(V_2) : \mathit{loc}$.

(xi) *If* $\Gamma \vdash V_1 \ \{r\} \ V_2 : \mathit{loc}$ *then* $\Gamma \vdash !V_1 \ \{r\} \ !V_2 : \mathit{int}$.

(xii) *If* $\Gamma \vdash V_1 \ \{r\} \ V_2 : \mathit{loc}$ *and* $\Gamma \vdash V_1' \ \{r\} \ V_2' : \mathit{int}$ *then* $\Gamma \vdash (V_1 := V_1') \ \{r\} \ (V_2 := V_2') : \mathit{unit}$.

(xiii) *If* $\Gamma \vdash V_1 \ \{r\} \ V_2 : \sigma \to \sigma'$ *and* $\Gamma \vdash V_1' \ \{r\} \ V_2' : \sigma$ *then* $\Gamma \vdash (V_1 V_1') \ \{r\} \ (V_2 V_2') : \sigma'$.

(xiv) *If* $\Gamma \vdash M_1 \ \{r\} \ M_2 : \sigma$ *and* $\Gamma, x : \sigma \vdash M_1' \ \{r\} \ M_2' : \sigma'$ *then* $\Gamma \vdash (\mathit{let}\, x = M_1 \,\mathit{in}\, M_1') \ \{r\} \ (\mathit{let}\, x = M_2 \,\mathit{in}\, M_2') : \sigma'$.

(xv) *If* $\Gamma, f : \sigma \to \sigma', x : \sigma \vdash M_1 \ \{r\} \ M_2 : \sigma'$ *then* $\Gamma \vdash (\mathit{rec}\, f(x).\, M_1) \ \{r\} \ (\mathit{rec}\, f(x).\, M_2) : \sigma \to \sigma'$.

Proof. As might be expected, many of the clauses here have similar proofs, with only four of them ((v), (x), (xiv) and (xv)) requiring individual attention. Moreover, there is no hard work: essentially all the proofs are compositions of properties proved earlier. First though we note some general points. Because each clause preserves Γ and makes no use of it, we may assume without loss of generality that all of its variables have already been substituted by $\{r\}$-related values, and so take $\Gamma = \emptyset$. We may similarly take just r rather than an extension $r' \rhd r$ in all clauses where Γ is the only source of free variables (*i.e.* (ii)–(xiii)). Lemma 4.4, which says that $\mathcal{V}_\sigma(r)$ and $\mathcal{E}_\sigma(r)$ coincide on values, allows us to move between the two relations and we do this silently throughout.

Cases (i)–(iv) all follow immediately from the definition of $\{r\}$ and $\mathcal{V}_\sigma(r)$. In case (v) we need to show that

$$(!\ell, !\ell) \in \mathcal{E}_{\mathit{int}}(r \otimes \mathit{id}_{\{\ell\}}) \quad \text{and} \quad \forall n \in \mathbb{Z}\,.\,(\ell := n, \ell := n) \in \mathcal{E}_{\mathit{unit}}(r \otimes \mathit{id}_{\{\ell\}})$$

for a given location ℓ not mentioned by r. We look at the first of these: the second is treated similarly. Suppose that $(s_1, s_2) \in (r \otimes \mathit{id}_{\{\ell\}} \otimes r')$ and $(K_1, K_2) \in \mathcal{K}_{\mathit{int}}(r \otimes \mathit{id}_{\{\ell\}} \otimes r')$ for some r'. We have to show

$$\langle s_1, K_1, !\ell\rangle \updownarrow \langle s_2, K_2, !\ell\rangle. \tag{4.7}$$

As $(s_1, s_2) \in (r \otimes \mathit{id}_{\{\ell\}} \otimes r')$ we know that $s_1(\ell) = s_2(\ell) = n$ for some integer n. Then

$$\langle s_i, K_i, !\ell\rangle \updownarrow \langle s_i, K_i, n\rangle \quad \text{for } i = 1, 2. \tag{4.8}$$

But $(n, n) \in \mathcal{V}_{\mathit{int}}(r \otimes \mathit{id}_{\{\ell\}} \otimes r')$ and as $(K_1, K_2) \in \mathcal{K}_{\mathit{int}}(r \otimes \mathit{id}_{\{\ell\}} \otimes r')$,

$$\langle s_1, K_1, n\rangle \updownarrow \langle s_2, K_2, n\rangle. \tag{4.9}$$

Combining (4.8) and (4.9) gives (4.7) as required.

Cases (vi), (vii) and (viii) are all alike and we look only at the first. We have

$$(V_1, V_2) \in \mathcal{E}_\sigma(r) \quad \text{and} \quad (V_1', V_2') \in \mathcal{E}_{\sigma'}(r), \tag{4.10}$$

and need to show $((V_1, V_1'), (V_2, V_2')) \in \mathcal{V}_{\sigma\times\sigma'}(r)$. This requires

$$\begin{array}{c}(\, fst(V_1, V_1'),\ fst(V_2, V_2')\,) \in \mathcal{E}_\sigma(r) \\ (\, snd(V_1, V_1'),\ snd(V_2, V_2')\,) \in \mathcal{E}_{\sigma'}(r).\end{array} \tag{4.11}$$

The following ciu-equivalences are all straightforward:

$$\begin{array}{ll} fst(V_1, V_2) \cong^{ciu} V_1 & fst(V_1', V_2') \cong^{ciu} V_1' \\ snd(V_1, V_2) \cong^{ciu} V_2 & snd(V_1', V_2') \cong^{ciu} V_2' \end{array} \tag{4.12}$$

and by Lemma 4.7 we can combine these with the logical relations of (4.10) to give (4.11) as required. Such use of ciu-equivalence is also the key to cases (vii) and (viii).

In case (ix) we are given $(V_1, V_2) \in \mathcal{V}_{\sigma\times\sigma'}(r)$ and need to show that $(fst(V_1), fst(V_2)) \in \mathcal{E}_\sigma(r)$ and $(snd(V_1), snd(V_2)) \in \mathcal{E}_{\sigma'}(r)$—but this is exactly the definition of $\mathcal{V}_{\sigma\times\sigma'}(r)$ on page 22. Cases (xi), (xii) and (xiii) use the definition of $\mathcal{V}(r)$ at locations and function types in exactly the same way.

For case (x) we again need to consider actual continuations. Suppose that $(V_1, V_2) \in \mathcal{V}_{int}(r)$, *i.e.* $V_1 = V_2 = n$ for some $n \in \mathbb{Z}$. For any $r' \rhd r$, $(s_1, s_2) \in r'$, and $(K_1, K_2) \in \mathcal{K}_{loc}(r')$ we need to show that

$$\langle s_1, K_1, ref(n)\rangle \updownarrow \langle s_2, K_2, ref(n)\rangle. \tag{4.13}$$

By considering a single reduction step[4] we have that

$$\langle s_i, K_i, ref(n)\rangle \updownarrow \langle s_i \otimes (\ell := n), K_i, \ell\rangle \quad \text{for } i = 1, 2 \text{ and } \ell \text{ fresh.} \tag{4.14}$$

Now $(s_1 \otimes (\ell{:=}n), s_2 \otimes (\ell{:=}n)) \in r' \otimes id_{\{\ell\}}$, by the weakening Lemma 4.3 we have $(K_1, K_2) \in \mathcal{K}_{loc}(r' \otimes id_{\{\ell\}})$, and from part (v) we know $(\ell, \ell) \in \mathcal{V}_{loc}(r' \otimes id_{\{\ell\}})$. Thus

$$\langle s_1 \otimes (\ell := n), K_1, \ell\rangle \updownarrow \langle s_2 \otimes (\ell := n), K_2, \ell\rangle \tag{4.15}$$

and this in combination with (4.14) gives (4.13) as required.

In part (xiv) we have the hypothesis that

$$(M_1, M_2) \in \mathcal{E}_\sigma(r) \quad \text{and} \quad x : \sigma \vdash M_1' \,\{r\}\, M_2' : \sigma'.$$

Consider $r' \rhd r$, $(s_1, s_2) \in r'$, and $(K_1, K_2) \in \mathcal{K}_{\sigma'}(r')$: we need to show

$$\langle s_1, K_1, let\, x = M_1\, in\, M_1'\rangle \updownarrow \langle s_2, K_2, let\, x = M_2\, in\, M_2'\rangle.$$

[4]Strictly speaking, the left-to-right implication in (4.14) relies upon the easily verified fact that the termination relation $\downarrow$ is invariant under bijective renamings of location constants.

Again, taking one reduction step gives

$$\langle s_i, K_i, \mathit{let}\, x = M_i\, \mathit{in}\, M_i' \rangle \updownarrow \langle s_i, K_i \circ (x)M_i', M_i \rangle \quad \text{for } i = 1, 2,$$

and as $(M_1, M_2) \in \mathcal{E}_\sigma(r)$ it is sufficient to prove that

$$(\, K_1 \circ (x)M_1',\ K_2 \circ (x)M_2' \,) \in \mathcal{K}_\sigma(r'). \tag{4.16}$$

Suppose that $r'' \rhd r'$, $(s_1', s_2') \in r''$, and $(V_1, V_2) \in \mathcal{V}_\sigma(r'')$; we now need

$$\langle s_1', K_1 \circ (x)M_1', V_1 \rangle \updownarrow \langle s_2', K_2 \circ (x)M_2', V_2 \rangle. \tag{4.17}$$

A single reduction step gives

$$\langle s_i', K_i \circ (x)M_i', V_i \rangle \updownarrow \langle s_i', K_i, M_i'[V_i/x] \rangle \quad \text{for } i = 1, 2. \tag{4.18}$$

Finally the hypothesis on M_i' and choice of V_i gives $(M_1'[V_1/x], M_2'[V_2/x]) \in \mathcal{E}_\sigma(r'')$, from which we get

$$\langle s_1', K_1, M_1'[V_1/x] \rangle \updownarrow \langle s_2', K_2, M_2'[V_2/x] \rangle. \tag{4.19}$$

Combining this with (4.18) gives (4.17) and hence (4.16) as required.

For case (xv), concerning recursively defined function values, it is no surprise that we turn to the Unwinding Theorem 3.2. The first step is to show that for non-recursive functions

$$x : \sigma \vdash M_1 \,\{r\}\, M_2 : \sigma' \ \Rightarrow\ \vdash (\lambda\, x.\, M_1)\, \{r\}\, (\lambda\, x.\, M_2) \tag{4.20}$$

which is done through the ciu-equivalence

$$(\lambda\, x.\, M)V \cong^{ciu} M[V/x]$$

in much the same way as case (vi). Now suppose more generally that we have

$$f : \sigma \to \sigma', x : \sigma \vdash M_1 \,\{r\}\, M_2 : \sigma'. \tag{4.21}$$

As in Section 3, consider the following progressive unwindings:

$$\Omega \stackrel{\text{def}}{=} (\mathit{rec}\, f(x).\, fx)() \qquad F_{i,n+1} \stackrel{\text{def}}{=} \lambda\, x.\, M_i[F_{i,n}/f]$$
$$F_{i,0} \stackrel{\text{def}}{=} \lambda\, x.\, \Omega \qquad F_{i,\omega} \stackrel{\text{def}}{=} \mathit{rec}\, f(x).\, M_i\,.$$

for $i = 1, 2$. We then prove in turn

$$(\Omega, \Omega) \in \mathcal{E}_{\sigma\to\sigma'}(r) \tag{4.22}$$
$$(F_{1,n}, F_{2,n}) \in \mathcal{E}_{\sigma\to\sigma'}(r) \tag{4.23}$$
$$(F_{1,\omega}, F_{2,\omega}) \in \mathcal{E}_{\sigma\to\sigma'}(r). \tag{4.24}$$

The first of these is straightforward as Ω never terminates in any context; this provides the base case to prove (4.23) by induction on n, using (4.20) and (4.21) at each step; and finally the Unwinding Theorem 3.2 allows us to deduce (4.24), which is exactly the desired result:

$$\vdash (\mathit{rec}\, f(x).\, M_1)\ \{r\}\ (\mathit{rec}\, f(x).\, M_2) : \sigma \to \sigma'.$$

What we are using here is that the $\mathcal{E}_\sigma(r)$ relations are *admissible*, in an appropriate syntactic variant of the usual notion on domains: if every finite approximation of some (M_1, M_2) is in $\mathcal{E}_\sigma(r)$, then so is (M_1, M_2) itself. □

Theorem 4.9 (Fundamental Property of the Logical Relation).

(i) *Contexts preserve the identity logical relation: if* $\Gamma \vdash M_1\ \{id_\omega\}\ M_2 : \sigma$, *then for any context* $C[-]$ *with* $fv(C) \subseteq \Gamma' \subseteq \Gamma$, $loc(C[-]) \subseteq \omega$, *and* $\Gamma' \vdash C[M_i] : \sigma'$ *for* $i = 1, 2$ *(so that the hole '*$-$*' occurs in* $C[-]$ *within the scope of binding occurrences of the variables in* $\Gamma \setminus \Gamma'$*), it is the case that* $\Gamma' \vdash C[M_1]\ \{id_\omega\}\ C[M_2] : \sigma'$.

(ii) *The identity logical relation is reflexive: if* $M \in \mathit{Exp}_\sigma(\Gamma)$ *with* $loc(M) \subseteq \omega$, *then* $\Gamma \vdash M\ \{id_\omega\}\ M : \sigma$.

Proof. Part (i) is proved by induction on the structure of $C[-]$, using Proposition 4.8. Part (ii) is then the special case when $C[-]$ has no occurrences of the hole '$-$' at all, and is just an ordinary expression M. □

The following result draws together all the relations we have defined between ReFS expressions. It includes the 'ciu' theorem of Mason and Talcott (1992b).

Theorem 4.10 (Operational Extensionality). *The logical relation* $\{id_\omega\}$, *contextual equivalence* $\cong$, *and ciu-equivalence* $\cong^{ciu}$ *all coincide: for any* $M_1, M_2 \in \mathit{Exp}_\sigma(\Gamma)$

$$\Gamma \vdash M_1\ \{id_\omega\}\ M_2 \ \Leftrightarrow\ \Gamma \vdash M_1 \cong M_2 \ \Leftrightarrow\ \Gamma \vdash M_1 \cong^{ciu} M_2\,.$$

Proof. We show that each of the three relations entails the next, in rotation. First, that

$$\Gamma \vdash M_1\ \{id_\omega\}\ M_2 \ \Rightarrow\ \Gamma \vdash M_1 \cong M_2\,.$$

Suppose that M_1 and M_2 are identity related as shown and that $C[-]$ is some context with $\emptyset \vdash C[M_i] : \sigma'$ ($i = 1, 2$). Using the weakening Lemma 4.3 on $\Gamma \vdash M_1\ \{id_\omega\}\ M_2$ if necessary, we can assume without loss of generality that $loc(C[-]) \subseteq \omega$. Then by Theorem 4.9(i) we have $\emptyset \vdash C[M_1]\ \{id_\omega\}\ C[M_2]$, that is $(C[M_1], C[M_2]) \in \mathcal{E}_{\sigma'}(id_\omega)$. Given any state s with $dom(s) = \omega$ we know that $(s, s) \in id_\omega$; and $(\mathcal{I}d, \mathcal{I}d) \in \mathcal{K}_{\sigma'}(id_\omega)$ holds by definition of $\mathcal{K}_{\sigma'}(id_\omega)$. Therefore from the definition of $\mathcal{E}_{\sigma'}(id_\omega)$ we have

$$\langle s, \mathcal{I}d, C[M_1]\rangle \updownarrow \langle s, \mathcal{I}d, C[M_2]\rangle.$$

Equivalently (by Theorem 3.1),

$$s, C[M_1] \Downarrow \Leftrightarrow s, C[M_2] \Downarrow .$$

Since this holds for any suitable $C[-]$ and s, we have contextual equivalence of M_1 and M_2, as required.

Now for

$$\Gamma \vdash M_1 \cong M_2 \Rightarrow \Gamma \vdash M_1 \cong^{ciu} M_2 .$$

Consulting the note after Definition 4.6, we recall that ciu-equivalence for an expression M is entirely determined by the set of terminations

$$s, E[M[\vec{V}/\vec{x}]] \Downarrow$$

for states s, evaluation contexts $E[-]$ and values $\vec{V}$ to instantiate the free variables of M. This however corresponds exactly to termination in the context $E[(\lambda \vec{x}.\,(-))\vec{V}]$. As ciu-equivalence thus requires correspondence only in a subset of all contexts, it is clear that it is entailed by contextual equivalence, which requires agreement on all of them.

Finally,

$$\Gamma \vdash M_1 \cong^{ciu} M_2 \Rightarrow \Gamma \vdash M_1 \{id_\omega\} M_2 .$$

By Theorem 4.9(ii) we have $\Gamma \vdash M_1 \{id_\omega\} M_1$, and Lemma 4.7 lets us compose this with $\Gamma \vdash M_1 \cong^{ciu} M_2$ to obtain the logical relation $\Gamma \vdash M_1 \{id_\omega\} M_2$ that we desire. □

5 Examples

In this section we look at some practical applications of the parametric logical relation to proving ReFS contextual equivalences, via the Operational Extensionality Theorem 4.10.

Before giving applications that make overt use of the logical relation, let us recall (from Mason and Talcott 1992b, for example) that the coincidence of contextual equivalence with ciu-equivalence gives sufficient leverage to prove a range of basic contextual equivalences. This includes those ciu-equivalences that hold by virtue of the immediate evaluation behaviour of the terms involved, such as

$$(rec\, f(x).\, M)V \;\cong\; M[\,(rec\, f(x).\, M)/f,\, V/x\,].$$

It also includes equivalences incorporating ‘garbage collection’, in the manner of Mason’s ‘strong isomorphism’ (Mason 1986; Mason and Talcott 1991a), allowing us to ignore parts of the store that are unreachable.

Using logical relations

To prove specific contextual equivalences via the logical relation, we need to be able to show that pairs of expressions are indeed related. The next result gives a general technique for demonstrating $(M_1, M_2) \in \mathcal{E}_\sigma(r)$: if their evaluation preserves r and we can exhibit some 'local invariant' r'' that correctly captures the way M_1 and M_2 treat their local variables, then they are logically related.

Proposition 5.1 (Principle of Local Invariants). *Given a state relation r and expressions M_1, M_2, suppose that for all $(s_1, s_2) \in r$,*

$$s_1, M_1 \Downarrow V_1, s_1' \;\Rightarrow\; \exists r', s_2', V_2 \,.\, (s_1', s_2') \in (r \otimes r') \\ \& \; s_2, M_2 \Downarrow V_2, s_2' \;\&\; (V_1, V_2) \in \mathcal{E}_\sigma(r \otimes r')$$

and

$$s_2, M_2 \Downarrow V_2, s_2' \;\Rightarrow\; \exists r'', s_1', V_1 \,.\, (s_1', s_2') \in (r \otimes r'') \\ \& \; s_1, M_1 \Downarrow V_1, s_1' \;\&\; (V_1, V_2) \in \mathcal{E}_\sigma(r \otimes r'').$$

Then $(M_1, M_2) \in \mathcal{E}_\sigma(r)$. (We call the state relations r', r'' local invariants.*)*

Proof. To prove $(M_1, M_2) \in \mathcal{E}_\sigma(r)$ it suffices to show for all $(s_1, s_2) \in r$ and $(K_1, K_2) \in \mathcal{K}_\sigma(r)$ that $\langle s_1, K_1, M_1\rangle\!\downarrow$ if and only if $\langle s_2, K_2, M_2\rangle\!\downarrow$ (cf. the *Note* on page 22). By the symmetry of the assumptions, it suffices to prove just the forward direction of this bi-implication. So suppose $\langle s_1, K_1, M_1\rangle\!\downarrow$. Then by (3.3) for some s_1' and V,

$$s_1, M_1 \Downarrow V_1, s_1' \quad \text{and} \quad \langle s_1', K_1, V_1\rangle\!\downarrow.$$

So by hypothesis there are r', s_2' and V_2 such that $s_2, M_2 \Downarrow V_2, s_2'$, $(s_1', s_2') \in (r \otimes r')$, and $(V_1, V_2) \in \mathcal{E}_\sigma(r \otimes r')$. From Lemma 4.3 we have $(K_1, K_2) \in \mathcal{K}_\sigma(r \otimes r')$ and so $\langle s_1', K_1, V_1\rangle \updownarrow \langle s_2', K_2, V_2\rangle$. Hence $\langle s_2', K_2, V_2\rangle\!\downarrow$ and then the other direction of (3.3) gives $\langle s_2, K_2, M_2\rangle\!\downarrow$ as desired. Similarly, the second hypothesis implies that $\langle s_2, K_2, M_2\rangle\!\downarrow \;\Rightarrow\; \langle s_1, K_1, M_1\rangle\!\downarrow$ and so we have $(M_1, M_2) \in \mathcal{E}_\sigma(r)$ as required. □

This gives us a tool for proving instances of the logical relation. Conversely, if we have that two expressions are related, what can we deduce about them? To answer this requires a mild restriction on the state relations considered.

Definition 5.2. A state relation $r \in Rel(\omega_1, \omega_2)$ is *closed* if every non-element can be detected as such. That is, for each pair of states $(s_1, s_2) \in (Sta(\omega_1) \times Sta(\omega_2)) \setminus r$ there are continuations $(K_1, K_2) \in \mathcal{K}_{unit}(r)$ such that

$$\langle s_1, K_1, ()\rangle \not\updownarrow \langle s_2, K_2, ()\rangle$$

with one converging and the other diverging.

All the state relations we shall encounter are closed. In particular we have the following result.

Lemma 5.3. *If a state relation $r \in Rel(\omega_1, \omega_2)$ is bijective where defined*

$$(s_1, s_2) \in r \ \&\ (s_1, s_2') \in r \ \Rightarrow\ s_2 = s_2'$$
$$(s_1, s_2) \in r \ \&\ (s_1', s_2) \in r \ \Rightarrow\ s_1 = s_1'$$

then it is closed.

Proof. First we note that all states have finite domain, so for any state s we can write an expression $test_s$ such that $s', test_s \Downarrow$ precisely when s' is of the form $s \otimes s''$.

Now suppose $(s_1, s_2) \in (Sta(\omega_1) \times Sta(\omega_2)) \setminus r$. We seek r-related continuations that distinguish between the two. If there is no s_2' such that $(s_1, s_2') \in r$, then we choose

$$K_1 \stackrel{\text{def}}{=} \mathcal{I}d \circ (x)\, test_{s_1} \quad \text{and} \quad K_2 \stackrel{\text{def}}{=} \mathcal{I}d \circ (x)\Omega.$$

If on the other hand $(s_1, s_2') \in r$ holds for some s_2', then by assumption on r we must have $s_2' \neq s_2$ (since $(s_1, s_2) \notin r$) and we take

$$K_1 \stackrel{\text{def}}{=} \mathcal{I}d \circ (x)\, test_{s_1} \quad \text{and} \quad K_2 \stackrel{\text{def}}{=} \mathcal{I}d \circ (x)\, test_{s_2'}\,.$$

In either case we have $\langle s_1, K_1, ()\rangle\downarrow$ and $\langle s_2, K_2, ()\rangle\not\downarrow$, while these continuations agree on all state pairs in any $r \otimes r'$ and so $(K_1, K_2) \in \mathcal{K}_{unit}(r)$ as required. □

Proposition 5.4. *If $(M_1, M_2) \in \mathcal{E}_\sigma(r)$ and $(s_1, s_2) \in r$ then*

$$s_1, M_1 \Downarrow \ \Leftrightarrow\ s_2, M_2 \Downarrow\ .$$

If r is closed, then related expressions also preserve the state relation:

$$s_1, M_1 \Downarrow V_1, s_1' \quad \textit{and} \quad s_2, M_2 \Downarrow V_2, s_2'$$

implies there is r' such that $(s_1', s_2') \in (r \otimes r')$.

Proof. For the first part, consider the continuation $K = \mathcal{I}d \circ (x)()$ of type $(\sigma \circ\!\!\to unit)$. Clearly $(K, K) \in \mathcal{K}_\sigma(r)$ and so using (3.3) we can deduce

$$s_1, M_1 \Downarrow \ \Leftrightarrow\ \langle s_1, K, M_1\rangle\downarrow \ \Leftrightarrow\ \langle s_2, K, M_2\rangle\downarrow \ \Leftrightarrow\ s_2, M_2 \Downarrow$$

as required. For the second part, suppose that r is closed and that M_1 and M_2 evaluate as given. Taking $r' = \omega_1' \times \omega_2'$, where $\omega_i' = dom(s_i') \setminus dom(s_i)$ $(i = 1, 2)$, we need only show that $(s_1'|_{dom(r)}, s_2'|_{cod(r)}) \in r$. Since r is closed, if these states are not so related, then there is some $(K_1, K_2) \in \mathcal{K}_{unit}(r)$ that can detect this and hence for which

$$\langle s_1, K_1 \circ (x)(), M_1\rangle \not\Uparrow \langle s_2, K_2 \circ (x)(), M_2\rangle.$$

This however would contradict the assumption that $(M_1, M_2) \in \mathcal{E}_\sigma(r)$, and instead we must have $(s_1'|_{dom(r)}, s_2'|_{cod(r)}) \in r$ and hence $(s_1', s_2') \in (r \otimes r')$ as required. □

This provides a partial converse to Proposition 5.1 (at least as regards termination and treatment of store) which we will use in the higher-order profiling Example 5.8. At ground types we can do better and show that the resulting values will also be related. However, the converse of Proposition 5.1 does not hold in general. Example 5.9 gives a pair of logically related expressions where no choice of r'' will make V_1 and V_2 related. Thus as a method for demonstrating instances of the logical relation and hence contextual equivalence, the Principle of Local Invariants of Proposition 5.1 is not complete. Nevertheless, it does provide a powerful method for proving equivalences, as the examples in the following subsections show. We consider various idioms for using functions with local state and prove that they behave correctly, interacting properly with surrounding code that may itself use state and higher order functions. The general approach is that we present contextual equivalences which express the desired behaviour of a program fragment, and then prove that these hold by using the logical relation. In all cases the crucial step is to choose the right local invariant that captures the way an expression is expected to use its local store.

Representation independence

Informally, it is clear that if two functions in ReFS have private local store that they use in different ways to compute the same result, then they should be contextually equivalent. One can use coincidence of contextual equivalence with ciu-equivalence to show that this is true for expressions which use local store only for temporary variables. However in ReFS it is also possible to write functions that rely on store remaining private from one invocation to the next. Logical relations can capture this notion of privacy through local invariants, and we give here two examples of how this can lead to proofs of contextual equivalence.

Consider the following expressions of Standard ML:

$$
\begin{array}{l}
\texttt{let val c} = \texttt{ref 1} \\
\quad \texttt{fun inc}\,() \;\; = (\texttt{c:=!c} + \texttt{1}) \\
\quad \texttt{fun test}\,() = (\texttt{!c} > \texttt{0}) \\
\texttt{in} \\
\quad (\texttt{inc}, \texttt{test}) \\
\texttt{end}
\end{array}
\quad \cong \quad
\begin{array}{l}
\texttt{let} \\
\quad \texttt{fun skip}\,() \;\; = () \\
\quad \texttt{fun test}'\,() = \texttt{true} \\
\texttt{in} \\
\quad (\texttt{skip}, \texttt{test}') \\
\texttt{end}.
\end{array}
$$

The first of these evaluates to a pair of functions sharing a common storage cell `c`; one function to increment `c`, and one to test its contents. However the test always returns true and the increment cannot be observed; so this expression has an equivalent simpler version which doesn't bother with the cell `c`. A corresponding example in ReFS is this:

Example 5.5.

$$
\begin{array}{rcl}
(\mathit{let}\, c = \mathit{ref}(1)\, \mathit{in}\, (\mathit{inc}\ c, \mathit{test}\ c)) & \cong & (\mathit{skip}, \mathit{test}') \\
 & & : (\mathit{unit} \to \mathit{unit}) \times (\mathit{unit} \to \mathit{bool})
\end{array}
\tag{5.1}
$$

where

$$inc \stackrel{\text{def}}{=} \lambda\, c.\, \lambda\, x.\, (c := (!c + 1)) \qquad skip \stackrel{\text{def}}{=} \lambda\, x.\, ()$$
$$test \stackrel{\text{def}}{=} \lambda\, c.\, \lambda\, x.\, (!c > 0) \qquad test' \stackrel{\text{def}}{=} \lambda\, x.\, true.$$

Proof. Although the internal action of these expressions is quite different, they are contextually equivalent because the value stored in cell c is always positive. This invariance property is expressed by the state relation

$$r \stackrel{\text{def}}{=} \{(s, ()) \mid s(\ell) > 0\} \in Rel(\{\ell\}, \emptyset).$$

Looking at the bodies of the functions inc and $skip$ we can show by Proposition 5.1 that

$$(\,(\ell := !\ell + 1),\ ()\,) \in \mathcal{E}_{unit}(r)$$

because both preserve r. Similarly for the $test$ functions

$$(\,(!\ell > 0),\ true\,) \in \mathcal{E}_{bool}(r)$$

because they give equal results provided that r holds. By Proposition 4.8(xv) lambda abstraction preserves these relations and we can then derive

$$(inc\ \ell, skip) \in \mathcal{E}_{unit \to unit}(r) \quad \text{and} \quad (test\ \ell, test') \in \mathcal{E}_{unit \to bool}(r).$$

Proposition 4.8(vi) now gives

$$(\,(inc\ \ell, test\ \ell),\ (skip, test')\,) \in \mathcal{E}_{(unit \to unit) \times (unit \to bool)}(r)$$

which are the results of evaluating either side of (5.1). Since $((\ell := 1), ()) \in r$, the corresponding states resulting from this evaluation are also related and so by the Principle of Local Invariants (Proposition 5.1)

$$(\,(let\ c = ref(1)\ in\ (inc\ c, test\ c)),\ (skip, test')\,) \in \mathcal{E}_{(unit \to unit) \times (unit \to bool)}(id_\emptyset).$$

The contextual equivalence (5.1) then follows by Operational Extensionality Theorem 4.10. □

The next example considers not private shared state, but more visible store used in two different but equivalent ways. Consider these two counters in Standard ML:

```
let val c = ref 0                    ≅  let val c = ref 0
    fun up x = (c := !c + x; !c)            fun down x = (c := !c - x; 0 - !c)
in                                       in
    up                                       down
end                                      end
```

which can be written in ReFS thus:

Example 5.6.

$$(\mathit{let}\, c = \mathit{ref}\,(0)\ \mathit{in}\ \mathit{up}\ c) \cong (\mathit{let}\, c = \mathit{ref}\,(0)\ \mathit{in}\ \mathit{down}\ c)$$

where

$$\mathit{up} \stackrel{\text{def}}{=} \lambda\, c.\, \lambda\, x.\, (c := (!c + x); !c) \qquad \mathit{down} \stackrel{\text{def}}{=} \lambda\, c.\, \lambda\, x.\, (c := (!c - x); 0 - !c).$$

Proof. Both of these functions maintain an accumulator, summing the arguments to successive calls and returning a running total. Internally though, the second function reverses signs throughout. The appropriate local invariant is the relation $r \stackrel{\text{def}}{=} \{(s_1, s_2) \mid s_1(\ell) = -s_2(\ell)\} \in \mathit{Rel}(\{\ell\}, \{\ell\})$. For this we have $(\mathit{up}\ \ell, \mathit{down}\ \ell) \in \mathcal{E}_{\mathit{int}\to\mathit{int}}(r)$ and the proof of contextual equivalence proceeds as in the previous example. □

In both of these examples, the local store is not in fact private: the functions do export a certain limited access to it, both for reading and writing. What is important though is that this access is certain to preserve the relevant invariant, no matter how a surrounding program uses it; and as long as the invariant holds, the results returned by the given functions always agree.

Memoisation

One practical use for local state is in the implementation of a *memo function*. This is a function that retains a cache of past results in order to assist future computations. Logical relations provide a means to show that the consistency of this cache is maintained, whatever the surrounding program.

Here we consider a higher-order memoisation function, that transforms any 'repeatable' function into a memo function. For simplicity, we only record a single argument/result pair, and take both to be integers. In Standard ML one might define this by:

```
fun memoise f = let val a = ref 0 and r = ref(f 0);
                    fun f' x =
                        ((if x = !a then () else (a := x; r := f x)); !r)
                in
                    f'
                end

memoise : (int → int) → (int → int)
```

The idea here is that `memoise` modifies function `f` by attaching two private cells, `a` and `r`, to hold the argument and result of its most recent invocation. The resulting function `f'` acts like `f`, except that when called with the same argument twice in succession it uses the cached result `!r`, saving recomputation. This can be written in ReFS as follows.

Example 5.7. Let

$$memoise \stackrel{\text{def}}{=} \lambda f.\, let\, a = ref(0); r = ref(f\,0) \quad in \quad \lambda x.\, ((if\ x = !a\ then\ ()\ else\ (a := x; r := fx)); !r)$$

We say that $F \in Val_{int\to int}$ *computes* some total function $\phi : \mathbb{Z} \to \mathbb{Z}$ if for each state s with $loc(F) \subseteq dom(s)$ and every $n \in \mathbb{Z}$,

$$s, Fn \Downarrow \phi(n), (s \otimes s_n)$$

for some s_n. Thus F may make use of local or global store, but its results are 'repeatable' in the sense that they do not depend on the global state s and s is unchanged at the end of evaluating the application of F to a numeral. We claim that such an F is suitable for memoisation:

$$memoise\, F \cong F. \tag{5.2}$$

In particular for each $n \in \mathbb{Z}$, $(memoise\, F)n$ computes the same integer as Fn, namely $\phi(n)$.

Proof. First note that

$$s, memoise\, F \Downarrow F', (s \otimes (\ell_a := 0) \otimes s_0 \otimes (\ell_r := \phi(0))) \tag{5.3}$$

where

$$F' \stackrel{\text{def}}{=} \lambda x.\, ((if\ x = !\ell_a\ then\ ()\ else\ \ell_a := x; \ell_r := Fx); !\ell_r)$$

is the 'memoised' version of F. A suitable local invariant is that locations ℓ_a and ℓ_r always hold a valid argument/result pair; which we express with the relation

$$r \stackrel{\text{def}}{=} \{(s, ()) \mid \phi(s(\ell_a)) = s(\ell_r)\} \in Rel(\omega_0 \ell_a \ell_r, \emptyset).$$

where $\omega_0 = dom(s_0)$ (which by α-conversion we can assume is disjoint from $\{\ell_a, \ell_r\}$). As before the Principle of Local Invariants (Proposition 5.1) shows that the bodies of F and F' are r-related, so we can use Proposition 4.8(xv) to obtain

$$(F', F) \in \mathcal{V}_{int\to int}(id_\omega \otimes r)$$

where $loc(F) \subseteq \omega$. Since the states $(\ell_a := 0; s_0; \ell_r := \phi(0))$ and $()$ are related by r, taking (5.3) we can apply Proposition 5.1 again to give

$$((memoise\, F), F) \in \mathcal{E}_{int\to int}(id_\omega).$$

The Operational Extensionality Theorem 4.10 then provides the desired contextual equivalence (5.2). □

We considered total functions $\phi : \mathbb{Z} \to \mathbb{Z}$ in this example only to simplify matters. Extending the definition of 'F computes ϕ in a repeatable fashion' to partial functions (when $\phi(n)$ is undefined Fn must diverge, and *vice versa*), (5.2) still holds *provided* we restrict attention to those ϕ for which $\phi(0)$ is defined, since $memoise$ initialises the cache using this value.

Note that this same memoisation function can be used repeatedly in a program, to give several memo functions each with their own local store. One memo function can even be used within another without interference. For example, if F and G are function abstractions computing ϕ and ψ, then the composition $F \circ G$ (definable in ReFS in the usual way) computes the composition of ϕ with ψ; and (5.2) together with the congruence properties of contextual equivalence imply

$$memoise((memoise\, F) \circ (memoise\, G)) \cong F \circ G$$

(although one of the memoisations is redundant).

Higher-order profiling

Next we consider the use of local state for *profiling* function use, *i.e.* recording the calls to a particular function as it is used within a larger program. We use contextual equivalence to express two important properties of the profiled function:

- it correctly counts the number of times it is called;
- the overall program is otherwise unaffected.

Both of these assertions are then proved using logical relations, although in this case we need to use Proposition 5.4 in addition to the Principle of Local Invariants (Proposition 5.1).

As with memoisation, a single higher order function can capture the whole operation of profiling. In Standard ML:

```
fun profile f = let val c = ref 0;
                    fun f' x = (c:=!c + 1; f x);
                    fun r () = !c
                in
                    (f', r)
                end
```

$$\texttt{profile} : (\sigma \to \sigma') \to ((\sigma \to \sigma') \times (\texttt{unit} \to \texttt{int}))$$

This `profile` takes any function `f` and returns an instrumented version `f'` together with a read operation `r`. Both `f'` and `r` share a private local counter `c`, incremented by each call to `f'` and read by means of `r()`. Otherwise `f'` behaves exactly as the original function `f`; which may include further side-effects on global or local store. The profiling operation is truly higher order, working with functions of all types; we could for example safely apply it to the `memoise` function described earlier.

In ReFS one can write this profiling functional as follows.

Example 5.8.

$$\mathit{profile} \stackrel{\text{def}}{=} \lambda f.\,(\mathit{let}\, c = \mathit{ref}(0)\, \mathit{in}\, (f', r)) \quad \text{where} \quad f' \stackrel{\text{def}}{=} \lambda x.\,(c := !c + 1; fx)$$
$$r \stackrel{\text{def}}{=} \lambda x.\,!c.$$

The fact the profiling correctly records function calls means that the following contextual equivalence between integer expressions:

$$\begin{array}{l} \mathit{let}\,(f', r) = \mathit{profile}\ F \\ \mathit{in} \\ \quad Pf'; FV; Qf'; \\ \quad r() + 1 \end{array} \cong \begin{array}{l} \mathit{let}\,(f', r) = \mathit{profile}\ F \\ \mathit{in} \\ \quad Pf'; f'V; Qf'; \\ \quad r() \end{array} \tag{5.4}$$

holds for *any* $F \in \mathit{Val}_{\sigma\to\sigma'}$, $P, Q \in \mathit{Val}_{(\sigma\to\sigma')\to \mathit{unit}}$, and $V \in \mathit{Val}_{\sigma}$.

Here the context $(Pf'; [-]V; Qf')$ represents a program using the instrumented function f'. Depending on whether its hole is filled with F or f', the final total $r()$ alters by 1. It is significant that the values P and Q have access only to f' in this context and cannot use r to read the current contents of the counter. More generally, the function f' on its own is indistinguishable from the original F:

$$\mathit{fst}(\mathit{profile}\ F) \cong F : \sigma \to \sigma' \tag{5.5}$$

In this contextual equivalence the read operator is thrown away, with *fst* selecting just the profiled function f'.

Proof. To demonstrate (5.4) and (5.5), look first at the evaluation of the expressions in (5.4). The computation is in three parts, followed by examination of the counter c using the read operation $r()$. Assume that the counter is bound to location ℓ and set

$$F' \stackrel{\text{def}}{=} \lambda x.\,(\ell := !\ell + 1; Fx). \tag{5.6}$$

Both sides of (5.4) begin with the same evaluation:

$$s \otimes (\ell := 0), PF' \Downarrow (), s' \otimes (\ell := n) \otimes s_1$$

for some $n \in \mathbb{Z}$. Thanks to the 'garbage collection' properties of contextual equivalence mentioned at the beginning of this section, the unreachable extra store s_1 need not concern us. The next step is the evaluation of a call to F or F':

$$\begin{array}{l} s' \otimes (\ell := n), FV \Downarrow V', s'' \otimes (\ell := n) \otimes s_2 \\ s' \otimes (\ell := n), F'V \Downarrow V', s'' \otimes (\ell := (n+1)) \otimes s_2 \end{array}$$

The only difference so far is the value stored at location ℓ. What is important now is that QF' preserves this but is otherwise unaffected. The appropriate local invariant is the relation

$$r \stackrel{\text{def}}{=} \{(s, s') \mid s'(\ell) = s(\ell) + 1\} \in \mathit{Rel}(\{\ell\}, \{\ell\}),$$

which is closed, by Lemma 5.3.

By the Fundamental Property of the logical relation (Theorem 4.9(ii)) we have that both F and Q are id_ω-related to themselves. Using Proposition 5.1 we can show directly that increment preserves r:

$$(\,(\ell := !\ell + 1),\ (\ell := !\ell + 1)\,) \in \mathcal{E}_{unit}(r).$$

Applying Proposition 4.8, we combine all these with the definition (5.6) of F' to deduce that the application QF' satisfies

$$(QF', QF') \in \mathcal{E}_{unit}(id_\omega \otimes r).$$

We know that the closed relation $(id_\omega \otimes r)$ holds before this application, and Proposition 5.4 now tells us that it also holds after it. Thus

$$\begin{array}{c} s'' \otimes (\ell := n), QF' \Downarrow (), s''' \otimes (\ell := n') \otimes s_3 \\ s'' \otimes (\ell := n+1), QF' \Downarrow (), s''' \otimes (\ell := n'+1) \otimes s_3 \end{array}$$

for some $n' \geq n$, s''', and s_3. The last computation for each alternative is then

$$\begin{array}{c} s''' \otimes (\ell := n'), (r() + 1) \Downarrow (n'+1), s''' \otimes (\ell := n') \\ s''' \otimes (\ell := n'+1), r() \Downarrow (n'+1), s''' \otimes (\ell := n'+1). \end{array}$$

The final states are $(id_\omega \otimes r)$-related and the returned values are equal; thus by the Principle of Local Invariants, the two original expressions are id_ω-related and the equivalence (5.4) follows.

The second equivalence (5.5), that F' on its own is indistinguishable from F, is more straightforward. We need to show that

$$(\,(let\, c = ref(0)\, in\, \lambda x.\, (c := !c + 1; fx)),\ F\,) \in \mathcal{E}_{\sigma\to\sigma'}(id_\omega)$$

and this follows by Proposition 5.1 from

$$(F', F) \in \mathcal{V}_{\sigma\to\sigma'}(id_\omega \otimes r)$$

where $r \stackrel{\text{def}}{=} \{(s, ()) \mid s \in Sta(\{\ell\})\}$. Thus the operation *profile* has exactly the behaviour we would expect. □

As with memoisation, we can apply *profile* many times to give several profiled functions, each with its own private counter. So when we write *profile* F, the function F may have subprocedures within it that are already recording profiles, without causing interference. The procedure can also be adapted to profile the recursive calls a function makes to itself. The proof in this case is no more complicated than before, thanks to the fact that logical relations are preserved by recursive function abstractions (Proposition 4.8(xv)).

A more intricate situation with shared store arises if we use a profiler that keeps the same global counter for each function that it modifies:

$$\begin{array}{rcl}
\mathtt{val}(\mathtt{g_profile}, \mathtt{g_read}) & = & \mathtt{let\ val\ c} = \mathtt{ref}\ 0; \\
& & \quad \mathtt{fun\ prof\ f\ x} = (\mathtt{c} := !\mathtt{c} + 1; \mathtt{f\ x}); \\
& & \quad \mathtt{fun\ read}\,() \ = !\mathtt{c} \\
& & \mathtt{in} \\
& & \quad (\mathtt{prof}, \mathtt{read}) \\
& & \mathtt{end}.
\end{array}$$

Analogues of the equivalences (5.4) and (5.5) can be given for this global profiler, and proved using logical relations.

Limitations of the Principle of Local Invariants

We give an example to show that the existence of a local state relation, as asserted in the hypotheses of the Principle of Local Invariants (Proposition 5.1), although sufficient, is not necessary in order for two expressions to be $\{id_w\}$-related and hence to be contextually equivalent.

Consider the following two second-order functions in Standard ML:

$$\begin{array}{rcl}
\mathtt{val\ awkward} & = & \mathtt{let\ val\ c} = \mathtt{ref}\ 0; \\
& & \quad \mathtt{fun\ upto_one\ f} = (\mathtt{c} := 1; \mathtt{f}(); !\mathtt{c}) \\
& & \mathtt{in} \\
& & \quad \mathtt{upto_one} \\
& & \mathtt{end} \\
\mathtt{val\ const_one} & = & \mathtt{fn\ f} \Rightarrow (\mathtt{f}(); 1) \\
\mathtt{awkward}, \mathtt{const_one} & : & (\mathtt{unit} \to \mathtt{unit}) \to \mathtt{int}.
\end{array}$$

Both of these evaluate to functions that take a command `f` as an argument, execute it and then return the value 1. The second achieves this in a straightforward manner, whereas the first achieves it in an awkward manner—through the function `upto_one` which fetches the return value 1 from private cell `c`. One expects `awkward` and `const_one` to be contextually equivalent because this cell, although it initially holds 0, is set to 1 before every inspection, and during the execution of `upto_one` in any context the function $\mathtt{f}()$ cannot reset `c` to 0. We shall turn this into a formal proof of equivalence below. First, let us express these functions in ReFS:

Example 5.9.

$$awkward \cong const\_one : (unit \to unit) \to int \tag{5.7}$$

where

$$\begin{array}{rcl}
awkward & \stackrel{\text{def}}{=} & let\ c = ref(0)\ in\ \lambda\, f.\, (c := 1; f(); !c) \\
const_one & \stackrel{\text{def}}{=} & \lambda\, f.\, (f(); 1).
\end{array}$$

To show (5.7) using Proposition 5.1 we would need $r \in Rel(\{\ell\}, \emptyset)$ satisfying

$$((\ell := 0),\ ()) \in r \tag{5.8}$$

$$\text{and} \quad (upto\_one, const\_one) \in \mathcal{E}(r). \tag{5.9}$$

where $upto\_one \stackrel{\text{def}}{=} \lambda f.(\ell := 1; f(); !\ell)$. From (5.8) we can deduce that $((\lambda x.(\ell := 0)),\ \lambda x.()) \in \mathcal{V}_{unit \to unit}(r)$. Using Proposition 4.8(xiii) to combine this with (5.9), we deduce

$$(upto\_one(\lambda x.(\ell := 0)),\ const\_one(\lambda x.())) \in \mathcal{E}_{int}(r).$$

But these expressions evaluate to give different results

$$(\ell := 0), upto\_one(\lambda x.(\ell := 0)) \Downarrow 0, (\ell := 0)$$
$$(), const\_one(\lambda x.()) \Downarrow 1, ().$$

and this is easily seen to be impossible for expressions related by $\mathcal{E}$ at a ground type. Thus we cannot have both (5.8) and (5.9) at the same time, and there is no way to prove contextual equivalence (5.7) through Proposition 5.1.

We are left in the situation that although (5.7) does hold, and by Theorem 4.10 the two expressions are therefore related by $\mathcal{E}(id_\emptyset)$, it seems hard to demonstrate this relation directly. The root of the problem is that the argument $(\lambda x.(\ell := 0))$, which causes $upto\_one$ to return the surprise value 0, cannot in fact be provided by any surrounding context. Note that the location ℓ is not entirely private, since its contents can be changed (from 0 to 1) by a use of $upto\_one$ in a context which knows nothing of ℓ (such as $[-](\lambda x.())$). (See Stark 1994, §5.4, Example 14 for a related example, this time of contextual inequivalence.)

This problem with $(\lambda x.(\ell := 0))$ resembles known subtleties of contextual equivalence in Algol due to the undefinability of so-called 'snapback' operations in the language (see Pitts 1997, Example 4.1—an example due to O'Hearn). However this particular example has no direct Algol equivalent, because it relies on the fact that $upto\_one$ can both change the state and return a value — such 'active integers' are intentionally excluded from Algol.

We conjecture that a more general form of logical relation can be used to demonstrate the equivalence (5.7) via a result like Proposition 5.1, if we use parameterising relations r with Kripke-style indexing to capture the one-way nature of state change. Here this would allow the following relation with two components:

$$r_1 \ \supset \ r_2 \qquad \text{where} \quad r_1 = \{(s, ()) \mid s(\ell) = 0 \text{ or } 1\}$$
$$r_2 = \{(s, ()) \mid s(\ell) = 1\}.$$

This is meant to express the fact that the value stored at location ℓ may progress from 0 to 1, but is then fixed. More complex examples of progressing state would require a more complex index structure.

Without such generalised logical relations, we can proceed only by brute force.

Proof of (5.7). By completeness of ciu-equivalence (Theorem 4.10) it is enough to consider evaluation in any continuation. In this case we derive the requirement that for any state s and continuation K with $\ell \notin dom(s) \supseteq loc(K)$

$$\langle s \otimes (\ell := 0), K, upto\_one \rangle \updownarrow \langle s, K, const\_one \rangle.$$

This is equivalent to showing that for any state s and expression M with $\ell \notin dom(s) \supseteq loc(M)$ and $fv(M) \subseteq \{g : (unit \to unit) \to int\}$

$$s \otimes (\ell := 0), M[upto\_one\,/g] \Downarrow \;\Leftrightarrow\; s, M[const\_one\,/g] \Downarrow . \tag{5.10}$$

This can be proved by computation induction, but we first need a suitably strong induction hypothesis. Define the predicate $\mathbb{P}(s, M)$ for states s and expressions M with $\ell \notin dom(s) \supseteq loc(M)$ according to

$$\begin{aligned}
\mathbb{P}(s, M) \stackrel{\text{def}}{\Leftrightarrow}\ & \text{There is a state } s' \text{ and a value } V \text{ with free variable } g \text{ such that} \\
& \text{1.} \qquad s, M[const_one\,/g] \Downarrow V[const_one\,/g], s' \\
& \text{and 2a.} \quad \forall n \in \mathbb{Z}.\ s \otimes (\ell := n), M[upto_one\,/g] \\
& \qquad\qquad \Downarrow V[upto_one\,/g], s' \otimes (\ell := 1) \\
& \text{or 2b.} \quad \forall n \in \mathbb{Z}.\ s \otimes (\ell := n), M[upto_one\,/g] \\
& \qquad\qquad \Downarrow V[upto_one\,/g], s' \otimes (\ell := n) \\
& \qquad\qquad \&\ \ s, M[\lambda f.\,\Omega/g] \Downarrow V[\lambda f.\,\Omega/g], s'.
\end{aligned}$$

This rather complex expression captures exactly the way that evaluations of $upto\_one$ and $const\_one$ correspond to each other in appropriate contexts. In particular, whenever $M[-/g]$ is evaluated, it either applies the function replacing g (case 2a), or it does not (case 2b).

The following properties hold of $\mathbb{P}(s, M)$:

$$\forall n \in \mathbb{Z}.\, (\mathbb{P}(s, M) \;\Leftrightarrow\; s \otimes (\ell := n), M[upto\_one\,/g] \Downarrow) \tag{5.11}$$

$$\mathbb{P}(s, M) \;\Leftrightarrow\; s, M[const\_one\,/g] \Downarrow . \tag{5.12}$$

The forward implications simply expand the definition of $\mathbb{P}(s, M)$. The reverse directions can be proved by induction on the height of proofs of the evaluations $s \otimes (\ell{:=}0), M[upto\_one\,/g] \Downarrow V', s'$ and $s, M[const\_one\,/g] \Downarrow V', s'$ respectively. All the details are routine: in particular every evaluation rule either creates one of the situations (2a) or (2b), or preserves an existing one.

These equivalences (5.11) and (5.12) immediately give (5.10), from which the original contextual equivalence (5.7) follows as indicated. □

6 Further topics

We have seen that for each type σ, two ReFS expressions are contextually equivalent if and only if they are $\{id_\omega\}$-related. Moreover, this characterisation

of contextual equivalence not only implies the Mason-Talcott 'ciu' theorem for ReFS, but also allows one to formalise various intuitive arguments about contextual equivalence based on invariant state relations. The examples in the previous section demonstrate that the logical relation provides a powerful method for establishing ReFS contextual equivalences. Clearly it is of interest to extend the techniques introduced here to larger fragments of Standard ML than that represented by the ReFS language. In particular we would like to be able to treat:

(a) recursively defined types (rather than the simple types of ReFS),

(b) references to data of any type (rather than the integer-valued references of ReFS), and

(c) no types at all!

(a) would enable one to tackle proofs of equivalence involving efficient implementation with pointers and arrays of data structures with purely functional behaviour. (One would probably want to consider abstract types at the same time.) (b) is of interest because of the connections between object-based programming and the use of storage for function and procedure values. By (c) we mean the kind of untyped imperative lambda calculus considered by Mason and Talcott (1991a) and others; its dynamics includes the phenomena that (a) and (b) introduce in a more disciplined way, and more besides.

Extension (a) takes us beyond the techniques used in this paper because we relied on the simple nature of ReFS types to define the relations $\mathcal{E}_\sigma(r)$, $\mathcal{K}_\sigma(r)$, and $\mathcal{V}_\sigma(r)$ *by induction on the structure of the type* σ (see Definition 4.2). In the non-simply typed case one could instead attempt to define these relations for all types simultaneously by solving a fixed point equation for a suitable operator on (families of) relations. The addition of algebraic data types (lists, trees, *etc.*) to ReFS could be accommodated in this way. However, general forms of data type declaration may have negative or mixed-variance occurrences of the type being defined. The property required of the logical relation at such a type takes the form of a fixed point equation for an operator that is non-monotone; so it is not immediately clear that it can be satisfied. For denotationally-based logical relations there are ways of overcoming this problem using the 'minimal invariance' property of recursively defined domains: see (Pitts 1994; Pitts 1996). Operational versions of the techniques in *loc. cit.* are possible and can be used to extend the results presented here to cover (a). (This suggestion has been taken up for a pure functional language in Birkedal and Harper 1997.)

Regarding (b), it is well known that the ability to store function values gives rise to more complex behaviour than storing only values of ground types. For example, it becomes possible to encode recursive function definitions. This is reflected in the denotational semantics of such storage by the need to solve a mixed-variance domain equation. In this respect the difficulties which must be overcome to define a suitable logical relation might seem to be similar to those in case (a). However,

there are further complications. In ReFS any dynamically created state has 'support' disjoint from the existing global state: the definition of the logical relation exploits this fact in the use it makes of the smash product $r \otimes r'$ of state-relations. This disjointness of support breaks down with (b), since a global location can get updated with a function value involving a freshly created location. (Consider for example $\mathtt{let\,val\,b} = \mathtt{ref}(\mathtt{fn\,x} \Rightarrow 0)\,\mathtt{in\,a} := (\mathtt{fn\,x} \Rightarrow (!\mathtt{b})0)\,\mathtt{end}$, where a is some previously declared identifier of type $(\mathtt{int} \to \mathtt{int})\,\mathtt{ref}$.) Therefore the way the parameterisation of the logical relation treats dynamic allocation is more complicated in the presence of (b) and it remains to be seen if the techniques of this paper extend to cover this case.

Even for ReFS itself, it is possible to consider more refined versions of the parameterisation. For example one can consider relations on the flat complete partial order of states rather than on the set of states. Building on work of O'Hearn and Reynolds (1996), Pitts (1997, Example 4.1) shows how this small alteration helps with operational reasoning about divergence in Idealised Algol. It seems likely that it would be similarly useful for ReFS.

Of course the above list of enhancements is hardly complete: one might well want to consider I/O effects, or exception mechanisms, for example. When adding language features, we also need to keep in mind the feasibility of the proof method. The strength of the technique presented here lies as much in its usability as in its theoretical power. Although one may need to develop operationally-based analogues of some rather sophisticated methods from domain theory in order to define a logical relation (and establish its Fundamental Property) for function references and recursive datatypes, these technicalities do not necessarily complicate the *use* of logical relations to prove contextual equivalence. We can already see this in the treatment of recursion. The proof of Proposition 4.8(xv) is not straightforward, but its statement is simple and its use unrestricted: the logical relation is preserved by any recursive function abstraction. Our aim is to further increase the power of operationally-based logical relations without compromising their ease of use.

The operationally based logical relation we have presented here seems to provide a convenient and reasonably powerful method for proving contextual equivalences between functions with local store. Once a correct relation between states has been identified, verifying equivalence involves routine calculations with the structural operational semantics of the language. However, the examples we have given are all small-scale. It would be interesting to investigate machine-assistance for proofs using our methods. Note that we do not necessarily have to implement the proof that logical relations imply contextual equivalence; what might benefit from machine-assistance is the demonstration, in the case of large programs, that two expressions are $\{id_\omega\}$-related.

References

Birkedal, L. and R. Harper (1997). Relational interpretation of recursive types in an operational setting (Summary). In *Proc. TACS'97*, Lecture Notes in Computer Science. Springer-Verlag, Berlin. To appear.

Felleisen, M. and D. P. Friedman (1986). Control operators, the SECD-machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pp. 193–217. North Holland.

Harper, R., B. F. Duba, and D. MacQueen (1993). Typing first-class continuations in ML. *Journal of Functional Programming 3*(4), 465–484.

Harper, R. and C. Stone (1996). A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU–CS–96–136R, Carnegie Mellon University, Pittsburgh, PA.

Honsell, F., I. A. Mason, S. F. Smith, and C. L. Talcott (1995). A variable typed logic of effects. *Information and Computation 119*(1), 55–90.

Hughes, R. J. M. (1989). Why functional programming matters. *The Computer Journal 32*(2), 98–107.

Mason, I. A. (1986). *The Semantics of Destructive Lisp*. Ph. D. thesis, Stanford University. Also published as CSLI Lecture Notes Number 5, Center for the Study of Language and Information, Stanford University.

Mason, I. A., S. F. Smith, and C. L. Talcott (1996). From operational semantics to domain theory. *Information and Computation 128*(1), 26–47.

Mason, I. A. and C. L. Talcott (1991a). Equivalence in functional languages with effects. *Journal of Functional Programming 1*, 287–327.

Mason, I. A. and C. L. Talcott (1991b). Program transformations for configuring components. In *PEPM '91: Proceedings of the ACM/IFIP Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ACM SIGPLAN Notices 26(9), pp. 297–308.

Mason, I. A. and C. L. Talcott (1992a). Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science 105*, 167–215.

Mason, I. A. and C. L. Talcott (1992b). References, local variables and operational reasoning. In *Proceedings of the 7th Annual Symposium on Logic in Computer Science*, pp. 186–197. IEEE Computer Society Press.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press.

O'Hearn, P. W. and J. C. Reynolds (1996, April). From Algol to polymorphic linear lambda-calculus. Draft version, 46 pp.

O'Hearn, P. W. and R. D. Tennent (1993). Relational parametricity and local variables. In *20th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 171–184. ACM Press.

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

Paulson, L. C. (1991). *ML for the Working Programmer*. Cambridge University Press.

Pitts, A. M. (1994). Computational adequacy via 'mixed' inductive definitions. In *Proc. MFPS'93, New Orleans, LA, USA, April 1993*, Volume 802 of *Lecture Notes in Computer Science*, pp. 72–82. Springer-Verlag, Berlin.

Pitts, A. M. (1996). Relational properties of domains. *Information and Computation 127*, 66–90.

Pitts, A. M. (1997). Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent (Eds.), *Algol-Like Languages*, Volume 2, Chapter 17, pp. 173–193. Birkhauser. First appeared in *Proc. LICS'96*, pp 152–163, IEEE Computer Society Press, 1996.

Pitts, A. M. and I. D. B. Stark (1993). Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. MFCS'93, Gdańsk, 1993*, Volume 711 of *Lecture Notes in Computer Science*, pp. 122–141. Springer-Verlag, Berlin.

Plotkin, G. D. (1973). Lambda-definability and logical relations. Memorandum SAI-RM-4, School of Artificial Intelligence, University of Edinburgh.

Plotkin, G. D. (1980). Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley (Eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 363–373. Academic Press.

Reynolds, J. C. (1981). The essence of Algol. In J. W. de Bakker and J. C. van Vliet (Eds.), *Algorithmic Languages. Proceedings of the International Symposium on Algorithmic Languages*, pp. 345–372. North-Holland, Amsterdam.

Reynolds, J. C. (1982). Idealized Algol and its specification logic. In D. Néel (Ed.), *Tools and Notions for Program Construction*, pp. 121–161. Cambridge University Press.

Reynolds, J. C. (1983). Types, abstraction and parametric polymorphism. In R. E. A. Mason (Ed.), *Information Processing 83*, pp. 513–523. North-Holland, Amsterdam.

Seiber, K. (1995). Full abstraction for the second order subset of an ALGOL-like language. Technical Report A 04/95, Fach. Informatik, Univ. des Saarlandes, Saarbrücken, Germany.

Stark, I. D. B. (1994). *Names and Higher-Order Functions*. Ph. D. thesis, University of Cambridge. Also published as Technical Report 363, University of Cambridge Computer Laboratory, April 1995.

Talcott, C. (1997). Reasoning about functions with effects. In this volume.