Operational Semantics and Program Equivalence

Andrew M. Pitts

Cambridge University Computer Laboratory
Cambridge CB2 3QG, UK
Andrew.Pitts@cl.cam.ac.uk

Abstract. This tutorial paper discusses a particular style of operational semantics that enables one to give a 'syntax-directed' inductive definition of termination which is very useful for reasoning about operational equivalence of programs. We restrict attention to contextual equivalence of expressions in the ML family of programming languages, concentrating on functions involving local state. A brief tour of structural operational semantics culminates in a structural definition of termination via an abstract machine using 'frame stacks'. Applications of this to reasoning about contextual equivalence are given.

Table of Contents

1 Introduction ... 378
2 Functions with Local State ... 380
3 Contextual Equivalence ... 383
4 Structural Operational Semantics ... 386
5 Applications of the Abstract Machine Semantics ... 389
6 Conclusion ... 397
A Appendix: A Fragment of ML ... 397
  A.1 Types ... 397
  A.2 Expressions ... 398
  A.3 Evaluation Relation ... 399
  A.4 Type Assignment Relation ... 401
  A.5 Transition Relation ... 404
  A.6 An Abstract Machine ... 405
  A.7 A Structurally Inductive Termination Relation ... 407
B Exercises ... 409
C List of Notation ... 410

1 Introduction

The various approaches to giving meanings to programming languages fall broadly into three categories: denotational, axiomatic and operational. In a denotational semantics the meaning of programs is defined abstractly using elements of some suitable mathematical structure; in an axiomatic semantics, meaning is

G. Barthe et al. (Eds.): Applied Semantics, LNCS 2395, pp. 378–412, 2002.

defined via some logic of program properties; and in an operational semantics it is defined by specifying the behaviour of programs during execution. Operational semantics used to be regarded as less useful than the other two approaches for many purposes, because it tends to be quite concrete, with important general properties of a programming language obscured by a low-level description of how program execution takes place. The situation changed with the development of a *structural* approach to operational semantics initiated by Plotkin, Milner, Kahn, and others. Structural operational semantics is now widely used for specifying and reasoning about the semantics of programs.

In this tutorial paper I will concentrate upon the use of structural operational semantics for reasoning about program properties. More specifically, I will look at operationally-based proof techniques for *contextual equivalence* of programs (or fragments of programs) in the ML language—or rather, in a core language with function and reference types that is common to the various languages in the ML family, such as Standard ML [9] and Caml [5][1]. ML is a *functional* programming language because it treats 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. It is also a *procedural* language because it permits the use of references (or 'cells', or 'locations') for storing values: references can be declared locally in functions and then created dynamically and their contents read and updated as function applications are evaluated. Although this mix of (call-by-value) higher order functions with local, dynamically allocated state is conveniently expressive, there are many subtle properties of such functions up to contextual equivalence. The traditional methods of denotational semantics do not capture these subtleties very well—domain-based models either tend to be far from 'fully abstract', or very complicated, or both. Consequently a sort of 'back to basics' movement has arisen that attempts to develop theories of program equivalence for high-level languages based directly on operational semantics (see [11] for some of the literature).

There are several different styles of structural operational semantics (which I will briefly survey). However, I will try to show that one particular and possibly unfamiliar approach to structural operational semantics using a 'frame stack' formalism—derived from the approach of Wright and Felleisen [18] and used in the redefinition of ML by Harper and Stone [6]—provides a more convenient basis for developing properties of contextual equivalence of programs than does the evaluation (or 'natural', or 'big-step') semantics used in the official definition of Standard ML [9].

Further Reading. Most of the examples and technical results in this paper to do with operational properties of ML functions with local references are covered in more detail in the paper [15] written jointly with Ian Stark. More recent work on this topic includes the use of labelled transition systems and bisimulations by Jeffrey and Rathke [7]; and the use by Aboul-Hosn and Hannan of static restric-

[1] I will use the concrete syntax of Caml.

tions on local state in functions to give a more tractable theory of equivalence [1]. The use of logical relations based on abstract machine semantics to analyse other programming language features, such as polymorphism, is developed in [13, 14, 3]; see also [2] and [6].

Exercises. Some exercises are given in Appendix B.

Notation. A list of the notations used in this paper is given in Appendix C.

Acknowledgement. I am grateful to members of the audience of the lectures on which this paper is based for their lively feedback; and to an anonymous referee of the original version of this paper, whose detailed comments helped to improve the presentation.

2 Functions with Local State

Consider the following two Caml expressions p and m:

$$p \triangleq \texttt{let } a \texttt{ = ref 0 in} \quad \texttt{fun(}x \texttt{ : int) -> (}a \texttt{ := !}a \texttt{ + } x \texttt{ ; !}a\texttt{)} \tag{1}$$

$$m \triangleq \texttt{let } b \texttt{ = ref 0 in} \quad \texttt{fun(}y \texttt{ : int) -> (}b \texttt{ := !}b \texttt{ - } y \texttt{ ; 0 - !}b\texttt{)} \tag{2}$$

I claim that these Caml expressions (of type `int -> int`) are semantically equivalent, in the sense that we can use them interchangeably in any place in an ML program that expects an expression of type `int -> int` without affecting the overall behaviour of the program. Such notions of equivalence of programs go by the name of *contextual equivalence*. Here is an informal definition of this notion, that holds good for any particular kind of programming language:

> Two phrases of a programming language are *contextually equivalent* if any occurrences of the first phrase in a complete program can be replaced by the second phrase without affecting the observable results of executing the program.

This kind of program equivalence is also known as *operational*, or *observational* equivalence. To be more precise about it we have to define, for the programming language that concerns us, what we mean by a 'complete program' and by the 'observable results' of executing it. In fact different choices can be made for these notions, leading to possibly different notions of contextual equivalence for the same programming language. We postpone more precise definitions until the next section. First, let us work with this informal definition and explore some of the subtleties of mixing ML's functional and 'stateful' features.

The intuitive reason why the expressions p and m are contextually equivalent is that the property

> 'the contents of b is the negative of the contents of a'

is an invariant that is true throughout the life-time of the two expressions: it is true when they are first evaluated to get functions of type `int -> int` (because `-!a` $=$ `-0` $=$ `0` $=$ `!b` at that point); and whenever those functions are applied to an argument, although there are side-effects on the contents of `a` and `b`, the truth of the property remains invariant. Moreover, because the property holds, the values returned by the two functions (the contents of `a` in one case and the negative of the contents of `b` in the other case) are equal. So even though the contents of `a` and `b` may be different, since the only way we can use ML programs to observe properties of these locations is via applications of the functions created by evaluating p and m, we will never detect a difference between these two expressions.

That is the intuitive justification for the contextual equivalence of the expressions p and m. But it depends on assertions like

> 'the only way we can use ML programs to observe properties of these locations [the ones declared locally in the expressions] is via applications of the functions created by evaluating p and m'

whose validity is not immediately obvious. To rub home the point, let us look at another example.

$$f \triangleq \texttt{let } a \texttt{ = ref 0 in} \quad (3)$$
```
    let b = ref 0 in
    fun(x : int ref) -> if x == a then b else a
```

$$g \triangleq \texttt{let } c \texttt{ = ref 0 in} \quad (4)$$
```
    let d = ref 0 in
    fun(y : int ref) -> if y == d then d else c
```

Are these Caml expressions (of type `int ref -> int ref`) contextually equivalent? We might be led to think that they are equivalent, via the following informal reasoning, similar to that above. If we apply f to an argument ℓ, because we can always rename the bound identifiers `a` and `b` without changing the meaning of f [2], it seems that ℓ can never be equal to `a` and hence $f\,\ell$ is contextually equivalent to the private location `let a = ref 0 in a`. Similarly $g\,\ell$ should be contextually equivalent to the private location `let c = ref 0 in c`. But `let a = ref 0 in a` and `let c = ref 0 in c`, being α-convertible, are contextually equivalent. So f and g give contextually equivalent results when applied to any argument. If ML function expressions satisfied the usual extensionality principle for mathematical functions (see Fig. 1), then we could conclude that f and g are contextually equivalent.

The presence of dynamically created state in ML function expressions can cause them to not satisfy extensionality up to contextual equivalence. In particular the function expressions f and g defined in equations (3) and (4) are

[2] As we might hope, *α-convertible* expressions, i.e. ones differing only up to the names of their bound identifiers, turn out to be contextually equivalent.

'*Two functions* (defined on the same set of arguments) *are equal if they give equal results for each possible argument.*'

- True of mathematical functions (e.g. in set theory).
- False for ML function expressions in general.
- True for ML function expressions in canonical form (i.e. lambda abstractions), if we take 'equal' to mean contextually equivalent.
- True for pure functional programming languages (see [11]).
- True for languages with Algol-like block-structured local state (see [12]).

Fig. 1. Function Extensionality Principle

not contextually equivalent. To see this consider the following Caml interaction, where we observe a difference between the two expressions[3].

```
# let f = let a = ref 0 in let b = ref 0 in
          fun(x : int ref) -> if x == a then b else a ;;
val f : int ref -> int ref = <fun>
# let g = let c = ref 0 in
          let d = ref 0 in
          fun(y : int ref) -> if y == d then d else c ;;
val g : int ref -> int ref = <fun>
# let t = fun(h : int ref -> int ref) ->
          let z = ref 0 in h (h z) == h z ;;
val t : (int ref -> int ref) -> bool = <fun>
# t f ;;
- : bool = false
# t g ;;
- : bool = true
```

Thus the expression

$$t \triangleq \texttt{fun}(h : \texttt{int ref -> int ref}) \texttt{ ->} \\ \texttt{let } z \texttt{ = ref 0 in } h\,(h\,z) \texttt{ == } h\,z \tag{5}$$

has the property that $t\,f$ evaluates to `false` whereas $t\,g$ evaluates to `true`. (Why? If you are not familiar with the way ML expressions evaluate, read Sect. A.3 and then try Exercise B.1.) Thus f and g are not contextually equivalent expressions.

This example illustrates the fact that proving contextual *in*equivalence of two expressions is quite straightforward in principle—one just has to devise a suitable program that can use the expressions and give a different observable result with each. Much harder is the task of proving that expressions *are* contextually equivalent, since it appears that one has to consider all possible ways a program can use the expressions. For example, once we have given a proper

[3] I used the Objective Caml ⟨`www.ocaml.org`⟩ version 3.0 interpreter.

ML *evaluation relation* $\boxed{s, e \Rightarrow v, s'}$ where $\begin{cases} s = \text{initial } \textit{state} \\ e = \text{closed } \textit{expression} \text{ to be evaluated} \\ v = \text{resulting closed } \textit{canonical form} \\ s' = \text{final state} \end{cases}$

is inductively generated by rules *following the structure* of e; for example:

$$\frac{\begin{array}{l} s, e_1 \Rightarrow v_1, s' \\ s', e_2[v_1/\boldsymbol{x}] \Rightarrow v_2, s'' \end{array}}{s, \texttt{let } \boldsymbol{x} \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow v_2, s''}$$

(See Sect. A.3 for the full definition.)
Specifying semantics via such an evaluation relation is also known as *big-step* (anon), *natural* (Kahn [8]), or *relational* (Milner) semantics.

Fig. 2. ML Evaluation Relation (simplified, environment-free form)

definition of the notion of contextual equivalence, how do we give a rigorous proof that the expression in equation (1) is contextually equivalent to that in equation (2)? The rest of this paper introduces some methods for carrying out such proofs.

3 Contextual Equivalence

I hope the examples in Sect. 2, despite their artificiality, indicate that ML's combination of

recursively defined, higher order, call-by-value functions
+
statically scoped, dynamically created, mutable state

makes reasoning about properties of contextual equivalence of ML expressions very complicated. In fact even quite simple, general properties of contextual equivalence (such as the suitably restricted form of functional extensionality mentioned in Fig. 1) are hard to establish directly. To explain why, we need to look at the precise definition of ML contextual equivalence. To do that, I have to recall the form of operational semantics used in the Definition of Standard ML [9]: see Fig. 2. The fragment of ML we will work with is given in Sects A.1 and A.2. The full set of rules inductively defining the evaluation relation for this fragment of ML is given in Sect. A.3. In fact this is a simplified form of the evaluation relation actually used in the Definition of Standard ML. The latter has an *environment* component binding free identifiers to semantic values, whereas we will get by with this simpler form, in which the environment has been 'substituted in'. (Full ML also needs various auxiliary relations, for example to deal with exception-handling, but that will not concern us here.) One advantage of this 'substituted in' formulation is that the results of evaluation do not have to be specified as a separate syntactic category of 'semantic values', but rather are a subset of all the expressions, namely the ones in *canonical form* (see Sect. A.3); this simplifies the statement of some properties of the operational semantics (such as the Type Soundness Theorem A.1). A minor side-effect of this 'substituted in' formulation is that the names of *storage locations*[4], ℓ (drawn from a fixed, countably infinite set *Loc*), can occur in expressions explicitly—rather than implicitly via value identifiers bound to locations in the environment. Since we only consider locations for storing integers (see Fig. 6 in Sect. 5), we can take a memory *state* to be a finite function from the set *Loc* of names of storage locations to the set $\mathbb{Z}$ of integers.

Turning now to the definition of contextual equivalence, recall from Sect. 2 that we have to make precise two things:

- what constitutes a program
- what results of program execution we observe.

ML only evaluates expressions after they have been type-checked. So we take a *program* to be a well-typed expression with no free value identifiers: see Fig. 3. The rules inductively defining the type assignment relation for our fragment of ML are given in Sect. A.4. The *Type Soundness* Theorem A.1 in that section recalls an important relationship between typing and evaluation that we will use without comment from now on. (See Exercise B.2.)

ML *type assignment relation* $\boxed{\Gamma \vdash e : ty}$ where $\begin{cases} \Gamma = \textit{typing context} \\ e = \textit{expression} \text{ to be typed} \\ ty = \textit{type} \end{cases}$

is inductively generated by axioms and rules *following the structure* of e; for example:

$$\frac{\Gamma \vdash e_1 : ty_1 \quad \Gamma[x \mapsto ty_1] \vdash e_2 : ty_2 \quad x \notin dom(\Gamma)}{\Gamma \vdash (\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2) : ty_2}$$

(See Sect. A.4 for the full definition.)

The set of ML *programs* of type ty $\boxed{\mathrm{Prog}_{ty}}$ is defined to be $\{\, e \mid \emptyset \vdash e : ty \,\}$.

Fig. 3. ML programs are typed

The final ingredient needed for the definition of contextual equivalence is to specify which results of program execution we observe. In Sect. 2, I used the Objective Caml interpreter to observe a difference between the two expressions defined in equations (3) and (4). From this point of view, two results of evaluation, v, s and v', s' say, are observationally equal if $obs(v, s) = obs(v', s')$, where *obs* is the function recursively defined by

[4] or addresses, as the authors of [9] call them.

$$\left.\begin{aligned} obs(\mathtt{c}, s) &= \mathtt{c}, \quad \text{if } \mathtt{c} = \mathtt{true}, \mathtt{false}, \mathtt{n}, \mathtt{()} \\ obs(v_1 \;\mathtt{,}\; v_2, s) &= obs(v_1, s) \;\mathtt{,}\; obs(v_2, s) \\ obs(\mathtt{fun(}x \mathtt{\ :\ } ty\mathtt{)\ \text{->}\ } e, s) &= \mathtt{\text{<fun>}} \\ obs(\mathtt{fun\ } f \mathtt{\ =\ (}x \mathtt{\ :\ } ty\mathtt{)\ \text{->}\ } e, s) &= \mathtt{\text{<fun>}} \\ obs(\ell, s) &= \mathtt{\{contents=n\}}, \quad \text{if } (\ell \mapsto \mathtt{n}) \in s \end{aligned}\right\} \tag{6}$$

and which maps to a set of result expressions $\mathtt{r}$ given by

$$\begin{aligned} \mathtt{r} ::= {} & \mathtt{true} \\ & \mathtt{false} \\ & \mathtt{n} && (\mathtt{n} \in \mathbb{Z}) \\ & \mathtt{()} \\ & \mathtt{r\ ,\ r} \\ & \mathtt{\text{<fun>}} \\ & \mathtt{\{contents=n\}} && (\mathtt{n} \in \mathbb{Z}). \end{aligned}$$

But what if I had used a different interpreter—would it affect the notion of contextual equivalence? Probably not. Evidence for this is given by the fact that we can replace *obs* by a *constant function* and still get the same notion of contextual equivalence (see Exercise B.3). In other words, rather than observing particular things about the final results of evaluation, we can just as well observe the fact that there *is* some final result at all, i.e. observe *termination* of evaluation. This gives us the following definition of contextual equivalence.

Definition 3.1 (Contextual preorder/equivalence). Given $e_1, e_2 \in \mathrm{Prog}_{ty}$, define

$$\begin{aligned} e_1 =_{\mathrm{ctx}} e_2 : ty \;&\triangleq\; e_1 \leq_{\mathrm{ctx}} e_2 : ty \;\;\&\;\; e_2 \leq_{\mathrm{ctx}} e_1 : ty \\ e_1 \leq_{\mathrm{ctx}} e_2 : ty \;&\triangleq\; \forall \boldsymbol{x}, e, ty', s\,((\boldsymbol{x} : ty \vdash e : ty') \;\;\&\;\; s, e[e_1/\boldsymbol{x}]\Downarrow \;\supset\; s, e[e_2/\boldsymbol{x}]\Downarrow) \end{aligned}$$

where $s, e\Downarrow$ indicates *termination*:

$$s, e\Downarrow \;\triangleq\; \exists v, s'(s, e \Rightarrow v, s').$$

Remark 3.2 (Contexts). The program equivalence of Definition 3.1 is 'contextual' because it examines the termination properties of programs $e[e_i/\boldsymbol{x}]$ that contain occurrences of the expressions e_i being equated. If we replace e_i by a place-holder '$-$' (usually called a *hole*), then we get $e[-/\boldsymbol{x}]$, which is an example of what is usually called a (program) context. The programs e_i are *closed* expressions; for contextual equivalence of *open* expressions (ones possibly containing free identifiers) we would need to consider more general forms of context than $e[-/\boldsymbol{x}]$, namely ones in which the hole can occur within the scope of a binder, such as $\mathtt{fun(}y \mathtt{\ :\ } ty\mathtt{)\ \text{->}\ } -$. For simplicity, I have restricted attention to contextual equivalence of closed expressions, where we can use expressions with a free identifier in place of such general contexts without affecting $=_{\mathrm{ctx}}$.

Definition 3.1 is difficult to work with directly when it comes to reasoning about programs up to contextual equivalence. One problem is the quantification over all contexts $e[-/x]$. Thus to prove a property of e_1 up to contextual equivalence, it is not good enough just to know how e_1 evaluates—we have to prove termination properties for all uses $e[e_1/x]$ of it in a context $e[-/x]$. But in fact there is a more fundamental problem: *the definition of* $\Downarrow$ *is not syntax-directed.* For example, from the definition of the ML evaluation relation, we know that

$$s, \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \Downarrow$$

holds if

$$s', e_2[v_1/x] \Downarrow$$

where $s, e_1 \Rightarrow v_1, s'$; however, $e_2[v_1/x]$ is not built from subphrases of the original phrase $\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$.

Thus at first sight it seems that one cannot expect to prove properties involving termination (and in particular, properties of contextual equivalence) by induction on the structure of expressions. Indeed, in the literature one finds more complicated forms of induction used (involving measures of the size of contexts and the length of terminating sequences of reductions), often in combination with non-obvious strengthenings of induction hypotheses. However, we will see that it is possible to reformulate the operational semantics of ML to get a structurally inductive definition of termination that facilitates inductive reasoning about contextual equivalence, $=_{\mathrm{ctx}}$. To achieve that, we need to review the original approach to *structural operational semantics* of Plotkin [17] and subsequent refinements of it.

4 Structural Operational Semantics

The inductively defined ML evaluation relation (Fig. 2 and Sect. A.3) is an example of the S*tructural approach to* O*perational* S*emantics* (SOS) popularised by Plotkin [17]. SOS more closely reflects our intuitive understanding of various language constructs than did previous approaches to operational semantics using abstract machines, which tended to pull apart the syntax of those constructs and build non-intuitive auxiliary data structures. The word 'structural' refers to the fact that the rules of SOS inductive definitions are syntax-directed, i.e. follow the abstract, tree structure of the syntax. For example, the structure of the ML expression e determines what are the possible rules that can be used to deduce $s, e \Rightarrow v, s'$ from other valid instances of the ML evaluation relation. This is of great help when it comes to using an induction principle to prove properties of the inductively defined relation.

The SOS in [17] is formulated in terms of a *transition relation* (also known as a 'reduction', or 'small-step' relation). An appropriate transition relation for the fragment of ML we are considering takes the form of a binary relation between (state, expression)-pairs

$$(s\ ,\ e) \to (s'\ ,\ e')$$

that is inductively generated by rules following the structure of e. See Sect. A.5 for the complete definition. Theorem A.2 in that section sums up the relationship between the transition and evaluation relations.

The rules inductively defining such transition relations usually divide into two kinds: ones giving the basic steps of *reduction*, such as

$$\frac{v \text{ a canonical form}}{(s\ ,\ \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ v\ \texttt{in}\ e) \to (s\ ,\ e[v/\boldsymbol{x}])}$$

and ones for *simplification steps* that say how reductions may be performed within a context, such as

$$\frac{(s\ ,\ e_1) \to (s'\ ,\ e_1')}{(s\ ,\ \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ e_1\ \texttt{in}\ e_2) \to (s'\ ,\ \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ e_1'\ \texttt{in}\ e_2)}\ .$$

The latter can be more succinctly specified using the notion of *evaluation context* [18], as follows.

Lemma 4.1 (Felleisen-style presentation of $\to$). *$(s\ ,\ e) \to (s'\ ,\ e')$ holds if and only if $e = \mathcal{E}[r]$ and $e' = \mathcal{E}[r']$ for some* evaluation context *$\mathcal{E}$ and* basic reduction *$(s\ ,\ r) \to (s'\ ,\ r')$, where:*

- evaluation contexts *are expression contexts (i.e. syntax trees of expressions with one leaf replaced by a placeholder, or hole, denoted by $[-]$) that want to evaluate their hole; for the fragment of ML we are using, the evaluation contexts are given by*

$$\begin{array}{rll} \mathcal{E} ::= & [-] & \\ & \texttt{if}\ \mathcal{E}\ \texttt{then}\ e\ \texttt{else}\ e & \\ & \mathcal{E}\ \texttt{op}\ e & \text{for } \texttt{op} \in \{\texttt{=},\texttt{+},\texttt{-},\texttt{:=},\texttt{==},\texttt{;},\texttt{,}\} \\ & v\ \texttt{op}\ \mathcal{E} & \text{for } \texttt{op} \in \{\texttt{=},\texttt{+},\texttt{-},\texttt{:=},\texttt{==},\texttt{,}\} \\ & \texttt{op}\ \mathcal{E} & \text{for } \texttt{op} \in \{\texttt{!},\texttt{ref}\ ,\texttt{fst}\ ,\texttt{snd}\ \} \\ & \mathcal{E}\ e & \\ & v\ \mathcal{E} & \\ & \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ \mathcal{E}\ \texttt{in}\ e & \end{array} \tag{7}$$

  where e ranges over closed expressions (Sect. A.2) and v over closed expressions in canonical form (Sect. A.3);
- basic reductions $(s\ ,\ r) \to (s'\ ,\ r')$ *are the axioms in the Plotkin-style inductive definition of $\to$ (see Sect. A.5);*
- *$\mathcal{E}[r]$ denotes the expression resulting from replacing the 'hole' $[-]$ in $\mathcal{E}$ by the expression r.* □

The validity of Lemma 4.1 depends upon the fact (proof omitted) that every closed expression not in canonical form is uniquely of the form $\mathcal{E}[r]$ for some evaluation context $\mathcal{E}$ and some *redex* r, i.e. some expression appearing on the left-hand side of one of the basic reductions. So if we have a configuration $(s\ ,\ e)$ with e not in canonical form, then e is of the form $\mathcal{E}[r]$, there is a basic reduction $(s\ ,\ r) \to (s'\ ,\ r')$ and we make the next step of transition from $(s\ ,\ e)$ by replacing r by its corresponding *reduct* r' at the same time changing the state from s to s'.

Transitions $\boxed{\langle s \;,\, \mathcal{F}s \;,\, e\rangle \to \langle s' \;,\, \mathcal{F}s' \;,\, e'\rangle}$ where $\begin{cases} s, s' = \text{states} \\ \mathcal{F}s, \mathcal{F}s' = \textit{frame stacks} \\ e, e' = \text{closed expressions} \end{cases}$

are *defined by cases* (i.e. no induction), according to the structure of e and (then) $\mathcal{F}s$. For example:

$$\langle s \;,\, \mathcal{F}s \;,\, \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\rangle \to \langle s \;,\, \mathcal{F}s \circ (\texttt{let}\ x\ \texttt{=}\ [-]\ \texttt{in}\ e_2) \;,\, e_1\rangle$$
$$\langle s \;,\, \mathcal{F}s \circ (\texttt{let}\ x\ \texttt{=}\ [-]\ \texttt{in}\ e) \;,\, v\rangle \to \langle s \;,\, \mathcal{F}s \;,\, e[v/x]\rangle$$

(See Sect. A.6 for the full definition.)
Initial configurations of the abstract machine take the form $\langle s \;,\, \mathcal{I}d \;,\, e\rangle$ and *terminal* configurations take the form $\langle s \;,\, \mathcal{I}d \;,\, v\rangle$, where $\mathcal{I}d$ is the empty frame stack and v is a closed canonical form.

Fig. 4. An ML abstract machine

We can decompose any evaluation context into a nested composition of basic evaluation contexts, or so-called *evaluation frames*. In this way we arrive at a more elementary transition relation for ML—more elementary because the transition steps are defined by case analysis rather than by induction. This is shown in Fig. 4 and defined in detail in Sect. A.6 (see also [6] for a large-scale example of this style of SOS). The nested compositions of evaluation frames are usually called *frame stacks*. For the fragment of ML we are considering, they are given by:

$$\begin{array}{rll} \mathcal{F}s ::= & \mathcal{I}d & \text{empty} \\ & \mathcal{F}s \circ \mathcal{F} & \text{non-empty} \end{array}$$

and the evaluation frames $\mathcal{F}$ by:

$$\begin{array}{rll} \mathcal{F} ::= & \texttt{if}\ [-]\ \texttt{then}\ e\ \texttt{else}\ e & \\ & [-]\ \texttt{op}\ e & \text{for } \texttt{op} \in \{\texttt{=}, \texttt{+}, \texttt{-}, \texttt{:=}, \texttt{==}, \texttt{;}, \texttt{,}\} \\ & v\ \texttt{op}\ [-] & \text{for } \texttt{op} \in \{\texttt{=}, \texttt{+}, \texttt{-}, \texttt{:=}, \texttt{==}, \texttt{,}\} \\ & \texttt{op}\ [-] & \text{for } \texttt{op} \in \{\texttt{!}, \texttt{ref}, \texttt{fst}, \texttt{snd}\} \\ & [-]\ e & \\ & v\ [-] & \\ & \texttt{let}\ x\ \texttt{=}\ [-]\ \texttt{in}\ e\ . & \end{array}$$

(Just as not all expressions are well-typed, not all of the evaluation stacks in the above grammar are well typed. Typing for frame stacks is defined in Sect. 5.)

The relationship between the abstract machine steps and the evaluation relation of ML is summed up by Theorem A.3 in Sect. A.6. In particular we can express termination of evaluation in terms of termination of the abstract machine:

$$s, e \Downarrow \;\equiv\; \exists s', v\, (\langle s \;,\, \mathcal{I}d \;,\, e\rangle \to^* \langle s' \;,\, \mathcal{I}d \;,\, v\rangle)\,.$$

What one gains from the formulation of ML's operational semantics in terms of this abstract machine is the following simple, but key, observation.

The termination relation of the abstract machine

$$\searrow \triangleq \{ \langle s \,,\, \mathcal{F}s \,,\, e \rangle \mid \exists s', v\, (\langle s \,,\, \mathcal{F}s \,,\, e \rangle \to^* \langle s' \,,\, \mathcal{I}d \,,\, v \rangle) \}$$

has a direct, inductive definition following the structure of e *and* $\mathcal{F}s$: see Sect. A.7.

We have thus achieved the aim of reformulating the structural operational semantics of ML to get a structurally inductive characterisation of termination. Before outlining what can be done with this, it is perhaps helpful to contemplate the picture in Fig. 5, summing up the relationship between $\Downarrow$ and $\searrow$.

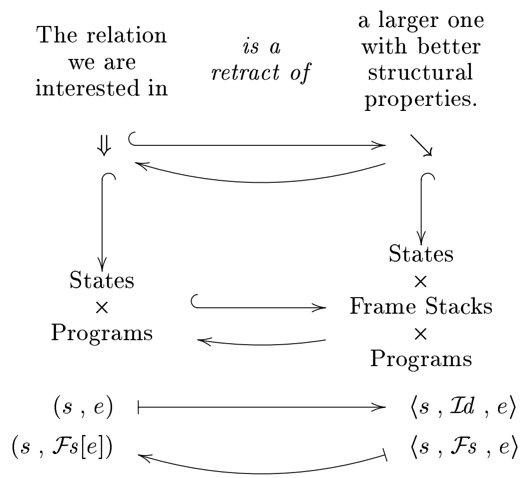

Fig. 5. The relationship between $\Downarrow$ and $\searrow$

5 Applications of the Abstract Machine Semantics

Recall the two ML expressions p and m defined by equations (1) and (2). In Sect. 2 it was claimed that they are contextually equivalent and some informal justification for this claim was given there. Now we can sketch how to turn that informal justification into a proper method of proof that uses a certain kind of binary 'logical relation' between ML expressions whose definition and properties depend on the abstract machine semantics of the previous section. Full details and several more examples of the use of this method can be found in [15][5]. The logical relation provides a method for proving contextual preorders

[5] In that work the logical relation is given in a symmetrical form that characterises contextual *equivalence*; here we use a one-sided version, a logical 'simulation' relation that characterises the contextual *preorder* (see Definition 3.1).

and equivalences that formalises intuitive uses of *invariant properties of local state*: given some binary relation between states, logically related expressions have the property that the change of state produced by evaluating them sends related states to related states (and produces logically related final values). This is made precise by property (I) in Theorem 5.2 below. One complication of ML compared with block-structured languages like Algol, is that local state is dynamically allocated: given an evaluation $s, e \Rightarrow v, s'$, the finite set $dom(s')$ of locations on which the final state s' is defined contains $dom(s)$, but may also contain other locations, ones that have been allocated during evaluation. Thus in formulating the notion of evaluation-invariant state-relations we have to take into account how the state on freshly allocated locations should be related. We accomplish this with the following definition.

Definition 5.1 (State-relations). We will refer to finite sets of locations as *worlds* (with a nod to the 'possible worlds' of Kripke semantics) and write them as $w, w_1, w_2, \ldots$ The set $\mathrm{St}(w)$ of *states in world* w is defined to be the set $\mathbb{Z}^w$ of integer-valued functions defined on w. The set $\mathrm{Prog}_{ty}(w)$ of *programs in world* w of type ty is defined to be $\left\{ e \in \mathrm{Prog}_{ty} \;\middle|\; loc(e) \subseteq w \right\}$. The set $\mathrm{Rel}(w_1, w_2)$ of *state-relations* between worlds w_1 and w_2 is defined to be the set of all non-empty[6] subsets of $\mathrm{St}(w_1) \times \mathrm{St}(w_2)$. Given two state-relations $r \in \mathrm{Rel}(w_1, w_2)$ and $r' \in \mathrm{Rel}(w_1', w_2')$ with $w_1 \cap w_1' = \emptyset$ and $w_2 \cap w_2' = \emptyset$, their *smash product* $r \otimes r' \in \mathrm{Rel}(w_1 \cup w_1', w_2 \cup w_2')$ is

$$r \otimes r' \triangleq \{ (s_1 s_1', s_2 s_2') \mid (s_1, s_2) \in r \;\&\; (s_1', s_2') \in r' \}$$

where if s and s' are states, then ss' is the state with $dom(ss') = dom(s) \cup dom(s')$ and for all $\ell \in dom(ss')$

$$(ss')(\ell) = \begin{cases} s'(\ell) \text{ if } \ell \in dom(s') \\ s(\ell) \;\text{ if } \ell \in dom(s) - dom(s'). \end{cases}$$

We say that a state relation $r' \in \mathrm{Rel}(w_1', w_2')$ *extends* a state relation $r \in \mathrm{Rel}(w_1, w_2)$, and write

$$r' \rhd r$$

if $r' = r \otimes r''$ for some r'' (so in particular $w_i \subseteq w_i'$ for $i = 1, 2$). (See Exercise B.5 for an alternative characterisation of the extension relation $\rhd$.)

Theorem 5.2 ('Logical' simulation relation between ML programs, parameterised by state-relations). *For each state-relation* $r \in \mathrm{Rel}(w_1, w_2)$ *we can define a relation*

$$e_1 \leq_r e_2 : ty \qquad (e_1 \in \mathrm{Prog}_{ty}(w_1), e_2 \in \mathrm{Prog}_{ty}(w_2))$$

(for each type ty), with the following properties:

[6] This non-emptiness condition is a technical convenience which, among other things, simplifies the definition of the logical relation (Definition 5.4) at ground types.

(I) ***The simulation property of $\le_r$:*** *to prove $e_1 \le_r e_2 : ty$, it suffices to show that whenever*

$$(s_1, s_2) \in r \quad \text{and} \quad s_1, e_1 \Rightarrow v_1, s'_1$$

then there exists $r' \rhd r$ and v_2, s'_2 such that

$$s_2, e_2 \Rightarrow v_2, s'_2, \quad v_1 \le_{r'} v_2 : ty \quad \text{and} \quad (s'_1, s'_2) \in r' \ .$$

(II) ***The extensionality properties of $\le_r$ on canonical forms:***

(i) *For $ty \in \{\mathtt{bool}, \mathtt{int}, \mathtt{unit}\}$, $v_1 \le_r v_2 : ty$ if and only if $v_1 = v_2$.*

(ii) *$v_1 \le_r v_2 : \mathtt{int\,ref}$ if and only if $!v_1 \le_r !v_2 : \mathtt{int}$ and for all $\mathtt{n} \in \mathbb{Z}$, $(v_1 := \mathtt{n}) \le_r (v_2 := \mathtt{n}) : \mathtt{unit}$.*

(iii) *$v_1 \le_r v_2 : ty_1 * ty_2$ if and only if $\mathtt{fst}\ v_1 \le_r \mathtt{fst}\ v_2 : ty_1$ and $\mathtt{snd}\ v_1 \le_r \mathtt{snd}\ v_2 : ty_2$.*

(iv) *$v_1 \le_r v_2 : ty_1 \mathtt{->} ty_2$ if and only if for all $r' \rhd r$ and all v'_1, v'_2*

$$v'_1 \le_{r'} v'_2 : ty_1 \ \supset \ v_1\, v'_1 \le_{r'} v_2\, v'_2 : ty_2.$$

(The last property is characteristic of (Kripke) logical relations [16, 10].)

(III) ***The relationship between $\le_r$ and contextual equivalence:*** *for all types ty, finite sets w of locations, and programs $e_1, e_2 \in \mathrm{Prog}_{ty}(w)$*

$$e_1 \le_{\mathrm{ctx}} e_2 : ty \quad \textit{iff} \quad e_1 \le_{id_w} e_2 : ty$$

where $id_w \in \mathrm{Rel}(w, w)$ is the identity *state-relation for w:*

$$id_w \triangleq \{\ (s, s) \mid s \in \mathrm{St}(w)\ \}\ .$$

Hence e_1 and e_2 are contextually equivalent if and only if both $e_1 \le_{id_w} e_2 : ty$ and $e_2 \le_{id_w} e_1 : ty$. □

We have two problems to discuss. First, why does the family of relations

$$- \le_r - : ty \qquad (r \in \mathrm{Rel}(w_1, w_2),\ w_1, w_2 \text{ finite subsets of } Loc,\ ty \text{ a type})$$

exist with the properties claimed in Theorem 5.2? Secondly, how do we use it to prove contextual equivalences like $p =_{\mathrm{ctx}} m : \mathtt{int->int}$ from Sect. 2? We address the second problem first. It is only when we get round to the first problem that we will see why the abstract machine semantics of the previous section is so useful.

Proof of the Contextual Equivalence of p and m. Consider the programs defined by equations (1) and (2). To prove $p =_{\mathrm{ctx}} m : \mathtt{int\ ->\ int}$, we have to show $p \le_{\mathrm{ctx}} m : \mathtt{int\ ->\ int}$ and $m \le_{\mathrm{ctx}} p : \mathtt{int\ ->\ int}$. We give the proof of the first contextual preorder; the argument for the second one is similar. Since $p, m \in \mathrm{Prog}_{\mathtt{int->int}}(\emptyset)$, by Theorem 5.2(III), to prove $p \le_{\mathrm{ctx}} m : \mathtt{int\ ->\ int}$ it suffices to prove $p \le_{id_\emptyset} m : \mathtt{int\ ->\ int}$. We do that by appealing to the simulation property of $\le_{id_\emptyset}$ given by Theorem 5.2(I). Note that $\mathrm{St}(\emptyset)$ contains only one element, namely the empty state $\emptyset$; hence the identity state-relation $id_\emptyset$ just contains the pair $(\emptyset, \emptyset)$ and we have to check the simulation property holds for this pair of states. So suppose

$$\emptyset, p \Rightarrow v, s.$$

It follows from the syntax-directed nature of the rules for evaluation in Sect. A.3 that v and s are uniquely determined up to the name of a freshly created location, call it ℓ_1:

$$v = \texttt{fun(}x \texttt{ : int) ->} \ \ell_1 \texttt{ := !}\ell_1 \texttt{ + } x \texttt{ ; !}\ell_1 \quad \text{and} \quad s = \{\ell_1 \mapsto \texttt{0}\}.$$

Choosing another new location ℓ_2, define

$$r \triangleq \{ (s_1, s_2) \mid s_1(\ell_1) = -s_2(\ell_2) \} \in \mathrm{Rel}(\{\ell_1\}, \{\ell_2\}).$$

Clearly $r \rhd id_\emptyset$ holds. Also the evaluation $\emptyset, m \Rightarrow v', s'$ holds with

$$v' = \texttt{fun(}y \texttt{ : int) ->} \ \ell_2 \texttt{ := !}\ell_2 \texttt{ - } y \texttt{ ; 0 - !}\ell_2 \quad \text{and} \quad s' = \{\ell_2 \mapsto \texttt{0}\}.$$

We certainly have $(s, s') \in r$, since $0 = -0$. So we just have to check that $v \leq_r v' : \texttt{int -> int}$. To do that we appeal to Theorem 5.2(II)(iv) and show for all $r' \rhd r$ and all $\texttt{n} \leq_{r'} \texttt{n}' : \texttt{int}$ that $v\,\texttt{n} \leq_{r'} v'\,\texttt{n}' : \texttt{int}$. By Theorem 5.2(II)(i), this amounts to showing

$$v\,\texttt{n} \leq_{r'} v'\,\texttt{n} : \texttt{int}, \quad \text{for all } \texttt{n} \in \mathbb{Z}. \tag{8}$$

We do this by once again appealing to the simulation property Theorem 5.2(I): given any $(s_1, s_2) \in r'$, since $r' \rhd r$ we have $(s_1\restriction_{\{\ell_1\}}, s_2\restriction_{\{\ell_2\}}) \in r$ and hence $s_1(\ell_1) = -s(\ell_2) = \texttt{k}$, say. Then

$$s_1, v\,\texttt{n} \Rightarrow \texttt{n}', s_1[\ell_1 \mapsto \texttt{n}'] \quad \text{and} \quad s_2, v'\,\texttt{n} \Rightarrow \texttt{n}', s_1[\ell_2 \mapsto -\texttt{n}']$$

with $\texttt{n}' = \texttt{k} + \texttt{n}$. From the definition of $r' \rhd r$ in Definition 5.1 it follows that $(s_1[\ell_1 \mapsto \texttt{n}'], s_1[\ell_2 \mapsto -\texttt{n}']) \in r'$; and $\texttt{n}' \leq_{r'} \texttt{n}' : \texttt{int}$ by Theorem 5.2(II)(i). So the simulation property does indeed imply equation (8) and hence we do have $v \leq_r v' : \texttt{int -> int}$, as required. □

Existence of the Logical Simulation Relation. We turn now to the problem of why the logical simulation relation described in Theorem 5.2 exists. Why can't we just take the simulation property (I) of the theorem as the *definition* of $- \leq_r - : ty$ at non-canonical expressions in terms of the logical simulation relation restricted to canonical expressions?—for then we could give a definition of the latter by induction on the structure of the type ty, using the extensionality properties in (II). The answer is that it seems impossible to connect such a version of $- \leq_r - : ty$ with contextual equivalence as in property (III) of the theorem, defeating the purpose of introducing these relations in the first place. The reason for this lies in the fact that we are dealing with a fragment of ML with *recursively defined* functions $\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e$ (and hence which is a Turing-powerful programming language, i.e. which can express all partial recursive functions from numbers to numbers)[7]. It turns out that each such recursively defined function is the least upper bound with respect to $\leq_{\mathrm{ctx}}$ of its finite unfoldings:

$$(\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e) \leq_{\mathrm{ctx}} g : ty\ \mathtt{->}\ ty' \;\equiv\; \forall n \geq 0\,(f_n \leq_{\mathrm{ctx}} g : ty\ \mathtt{->}\ ty') \quad (9)$$

where the expressions f_n are the 'finite unfoldings' of $\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e$, defined as follows:

$$\left.\begin{aligned} f_0 &\triangleq \mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ f\ x \\ f_{n+1} &\triangleq \mathtt{fun(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e[f_n/f] \end{aligned}\right\} \quad (10)$$

The least upper bound property in equation (9) follows immediately from the definition of $\leq_{\mathrm{ctx}}$ and the following 'Unwinding Theorem'.

Theorem 5.3 (An unwinding theorem). *Given*

$$f : ty\ \mathtt{->}\ ty', x : ty \vdash e : ty'$$

for each $n \geq 0$ define $f_n \in \mathrm{Prog}_{ty\mathtt{->}ty'}$ as in equation (10). Then for all

$$f : ty\ \mathtt{->}\ ty' \vdash e' : ty''$$

and all states s, it is the case that

$$s, e'[(\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e)/f]\Downarrow \quad\equiv\quad \exists n \geq 0\,(s, e'[f_n/f]\Downarrow)\ .$$

□

Proof. We can use the structurally inductive characterisation of termination afforded by Theorem A.4 to reduce the proof to a series of simple (if tedious) inductions. Writing f_ω for $\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e$, first show that

$$\langle s\ ,\ \mathcal{F}s[f_n/f]\ ,\ e'[f_n/f]\rangle\searrow \quad\supset\quad \langle s\ ,\ \mathcal{F}s[f_\omega/f]\ ,\ e'[f_\omega/f]\rangle\searrow$$

holds for all s, $\mathcal{F}s$, e' and n, by induction on the derivation of $\langle s\ ,\ \mathcal{F}s[f_n/f]\ ,\ e'[f_f/f]\rangle\searrow$ from the rules in Sect. A.7. Conversely, show for that

$$\langle s\ ,\ \mathcal{F}s[f_\omega/f]\ ,\ e'[f_\omega/f]\rangle\searrow \quad\supset\quad \exists n \geq 0(\langle s\ ,\ \mathcal{F}s[f_n/f]\ ,\ e'[f_n/f]\rangle\searrow)$$

holds all s, $\mathcal{F}s$ and e', by induction on the derivation of $\langle s\ ,\ \mathcal{F}s[f_\omega/f]\ ,\ e'[f_\omega/f]\rangle\searrow$. Doing this requires proving a sublemma to the effect that

$$\langle s\ ,\ \mathcal{F}s[f_n/f]\ ,\ e'[f_n/f]\rangle\searrow \quad\supset\quad \langle s\ ,\ \mathcal{F}s[f_{n+1}/f]\ ,\ e'[f_{n+1}/f]\rangle\searrow$$

which is done by induction on n, with the base case $n = 0$ proved by induction on the derivation of $\langle s\ ,\ \mathcal{F}s[f_0/f]\ ,\ e'[f_0/f]\rangle\searrow$. The unwinding theorem follows from these results by taking $\mathcal{F}s = \mathcal{I}d$ and applying Theorem A.4. □

[7] Compared with either Caml or Standard ML, $\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x : ty\mathtt{)}\ \mathtt{->}\ e$ is a non-standard canonical form; it is equivalent to the Caml expression $\mathtt{let\ rec}\ f\ \mathtt{=}\ \mathtt{(fun(}x : ty\mathtt{)}\ \mathtt{->}\ e\mathtt{)\ in}\ f$—see Sect. A.2.

If the logical relation $\leq_{id_w}$ is to coincide with $\leq_{\mathrm{ctx}}$ as in Theorem 5.2(III), it must have a property like (9). More generally, each $\leq_r$ should have a syntactic version of the 'admissibility' property that crops up in domain theory:

$$e'[(\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x : ty\mathtt{)}\ \mathtt{->}\ e)/f] \leq_r g{:}ty \;\equiv\; \forall n \geq 0\,(e'[f_n/f] \leq_r g{:}ty_1\mathtt{->}ty_2). \quad (11)$$

The problem with trying to use the simulation property of Theorem 5.2(I) as a definition of $\leq_r$ is that the existential quantification over extensions $r' \rhd r$ occurring in it makes it unlikely that equation (11) could be proved for that definition (although I do not have a specific counter-example to hand).

One can get round these difficulties by defining the logical simulation relation between expressions, $\leq_r$, in terms of a similar, auxiliary relation between frame stacks, $\mathrm{Stack}_{ty}(r)$; this in turn is defined using an auxiliary relation between canonical forms, $\mathrm{Val}_{ty}(r)$, that builds in the extensionality properties of Theorem 5.2(II). Since only well-typed expressions are considered, before giving the definitions of these auxiliary relations we need to define typing for (closed) frame stacks. We write $\vdash \mathcal{F}s : ty \multimap ty'$ to indicate that $\mathcal{F}s$ is a well-typed, closed frame stack taking an argument of type ty and returning a result of type ty'. This relation is inductively defined by the rules

$$\frac{}{\vdash \mathcal{I}d : ty \multimap ty} \qquad \frac{\begin{array}{l}\vdash \mathcal{F}s : ty' \multimap ty'' \\ x \notin fv(\mathcal{F}) \\ [x \mapsto ty] \vdash \mathcal{F}[x] : ty'\end{array}}{\vdash \mathcal{F}s \circ \mathcal{F} : ty \multimap ty''}$$

The set of *well-typed frame stacks taking an argument of type ty and only involving locations in the world w* is defined to be

$$\mathrm{Stack}_{ty}(w) \triangleq \{\, \mathcal{F}s \mid \exists ty' (\vdash \mathcal{F}s : ty \multimap ty') \,\}\,. \quad (12)$$

Definition 5.4 (A logical simulation relation). For all worlds w_1, w_2, state-relations $r \in \mathrm{Rel}(w_1, w_2)$ and types ty, we define binary relations between programs, frame stacks and canonical forms:

$$\begin{aligned} \leq_r &\subseteq \mathrm{Prog}_{ty}(w_1) \times \mathrm{Prog}_{ty}(w_2) \\ \mathrm{Stack}_{ty}(r) &\subseteq \mathrm{Stack}_{ty}(w_1) \times \mathrm{Stack}_{ty}(w_2) \\ \mathrm{Val}_{ty}(r) &\subseteq \mathrm{Val}_{ty}(w_1) \times \mathrm{Val}_{ty}(w_2). \end{aligned}$$

The relations between programs are defined in terms of those between frame stacks:

$$\begin{aligned} & e_1 \leq_r e_2 : ty \;\triangleq \\ & \quad \forall r' \rhd r, \forall (s_1', s_2') \in r', \forall (\mathcal{F}s_1, \mathcal{F}s_2) \in \mathrm{Stack}_{ty}(r') \\ & \qquad (\langle s_1'\,,\, \mathcal{F}s_1\,,\, e_1\rangle\searrow \;\supset\; \langle s_2'\,,\, \mathcal{F}s_2\,,\, e_2\rangle\searrow)\,. \end{aligned} \quad (13)$$

The relations between frame stacks are defined in terms of those between canonical forms:

$$\begin{aligned} & (\mathcal{F}s_1, \mathcal{F}s_2) \in \mathrm{Stack}_{ty}(r) \;\triangleq \\ & \quad \forall r' \rhd r, \forall (s_1', s_2') \in r', \forall (v_1, v_2) \in \mathrm{Val}_{ty}(r') \\ & \qquad (\langle s_1'\,,\, \mathcal{F}s_1\,,\, v_1\rangle\searrow \;\supset\; \langle s_2'\,,\, \mathcal{F}s_2\,,\, v_2\rangle\searrow)\,. \end{aligned} \quad (14)$$

The relations between canonical forms are defined by induction on the structure of the type ty, for all w_1, w_2 and r simultaneously:

$$
\begin{aligned}
(v_1, v_2) \in \mathrm{Val}_{\texttt{bool}}(r) &\equiv v_1 = v_2 && (15)\\
(v_1, v_2) \in \mathrm{Val}_{\texttt{int}}(r) &\equiv v_1 = v_2 && (16)\\
(\texttt{()}, \texttt{()}) \in \mathrm{Val}_{\texttt{unit}}(r) & && (17)\\
(v_1, v_2) \in \mathrm{Val}_{\texttt{int ref}}(r) &\equiv \texttt{!}v_1 \leq_r \texttt{!}v_2 : \texttt{int} \;\; \& && (18)\\
& \quad \forall \mathtt{n} \in \mathbb{Z}((v_1 \texttt{ := n}) \leq_r (v_2 \texttt{ := n}) : \texttt{unit}) &&\\
(v_1, v_2) \in \mathrm{Val}_{ty_1 \texttt{*} ty_2}(r) &\equiv \texttt{fst } v_1 \leq_r \texttt{fst } v_2 : ty_1 \;\; \& && (19)\\
& \quad \texttt{snd } v_1 \leq_r \texttt{snd } v_2 : ty_2 &&\\
(v_1, v_2) \in \mathrm{Val}_{ty_1 \texttt{->} ty_2}(r) &\equiv \forall r' \rhd r, \forall v_1', \forall v_2' && (20)\\
& \quad (v_1' \leq_{r'} v_2' : ty_1 \supset v_1\, v_1' \leq_{r'} v_2\, v_2' : ty_2).&&
\end{aligned}
$$

We extend the logical relation to open expressions via closing substitutions (of canonical forms for value identifiers). Thus given $\Gamma \vdash e : ty$ and $\Gamma \vdash e' : ty$ where $\Gamma = [\boldsymbol{x}_1 \mapsto ty_1, \ldots, \boldsymbol{x}_n \mapsto ty_n]$ say, and given $r \in \mathrm{Rel}(w_1, w_2)$ with $loc(e_i) \subseteq w_i$ for $i = 1, 2$, we define

$$\Gamma \vdash e \leq_r e' : ty \tag{21}$$

to mean that for all extensions $r' \rhd r$ and all related canonical forms $(v_i, v_i') \in \mathrm{Val}_{ty_i}(r')$ $(i = 1..n)$, it is the case that $e[\vec{v}/\vec{\boldsymbol{x}}] \leq_{r'} e'[\vec{v}'/\vec{\boldsymbol{x}}] : ty$ holds.

Proof of Theorem 5.2 (sketch). The proof that the relations $\leq_r$ of Definition 5.4 have all the properties required by Theorem 5.2 is quite involved. The details can be found in Sects 4 and 5 of [15]. Here is a guide to finding one's way through those details.

For part (I) of the theorem we use the following connection between evaluation and the structurally inductive termination relation (a generalisation of Theorem A.4):

$$\langle s \,,\, \mathcal{F}s \,,\, e \rangle\searrow \quad \equiv \quad \exists s', v \;\, (s, e \Rightarrow v, s' \;\;\&\;\; \langle s' \,,\, \mathcal{F}s \,,\, v\rangle\searrow) \;\; .$$

This, together with definitions (13) and (14), yield property (I) as in the proof of [15, Proposition 5.1].

Part (II) of the theorem follows from definitions (15)–(20) once we know that the restriction of the relation $- \leq_r - : ty$ to canonical forms coincides with $\mathrm{Val}_{ty}(r)(-,-)$; this is proved in [15, Lemma 4.4].

For part (III) of the theorem we have to establish the so-called "fundamental property" of the logical relation, namely that its extension to open expressions as in (21) is preserved by all the expression-forming constructs of the language. For example

$$
\begin{aligned}
&\text{if } \Gamma[\boldsymbol{f} \mapsto ty_1 \texttt{ -> } ty_2][\boldsymbol{x} \mapsto ty_1] \vdash e \leq_r e' : ty_2 \\
&\text{then } \Gamma \vdash (\texttt{fun } \boldsymbol{f} \texttt{ = (} \boldsymbol{x} \texttt{ : } ty_1\texttt{) -> } e) \leq_r \\
&\qquad\qquad (\texttt{fun } \boldsymbol{f} \texttt{ = (} \boldsymbol{x} \texttt{ : } ty_1\texttt{) -> } e') : ty_1 \texttt{ -> } ty_2 \;\; .
\end{aligned}
\tag{22}
$$

This property and similar ones for each of the other expression-forming constructs are proved in [15, Proposition 4.8]. In particular, the proof of (22) makes use of the Unwinding Theorem 5.3 to establish $\Gamma \vdash (\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty_1\mathtt{)}\ \mathtt{->}\ e) \leq_r (\mathtt{fun}\ f\ \mathtt{=}\ \mathtt{(}x\ \mathtt{:}\ ty_1\mathtt{)}\ \mathtt{->}\ e') : ty_1\ \mathtt{->}\ ty_2$ from the fact (proved by induction on n) that $\Gamma \vdash f_n \leq_r f'_n : ty_1\ \mathtt{->}\ ty_2$ holds for the finite approximations f_n, f'_n defined as in (10).

One immediate consequence of this fundamental property of the logical relation is that $\leq_{id_w}$ is a reflexive relation. Also, it is not hard to see that if two expressions are logically related and we change one of them up to the contextual preorder, we still have logically related expressions. Thus if $e_1 \leq_{\mathrm{ctx}} e_2 : ty$, since we have $e_1 \leq_{id_w} e_1 : ty$, we also have $e_1 \leq_{id_w} e_2 : ty$. This is one half of property (III). The other half also follows from the fundamental property. For if $e_1 \leq_{id_w} e_2 : ty$, then for any context $x : ty \vdash e : ty'$ (where without loss of generality we assume $loc(e) \subseteq w$), the fundamental property implies that $e[e_1/x] \leq_{id_w} e[e_2/x] : ty'$ holds. So if $s, e[e_1/x]\Downarrow$, then by Theorem A.4 $\langle s\ , \mathcal{I}d\ , e[e_1/x]\rangle\searrow$. Using the easily verified fact that $(\mathcal{I}d, \mathcal{I}d) \in \mathrm{Stack}_{ty}(id_w)$, it follows from $e[e_1/x] \leq_{id_w} e[e_2/x] : ty'$ and definition (13) that $\langle s\ , \mathcal{I}d\ , e[e_2/x]\rangle\searrow$ and hence that $s, e[e_2/x]\Downarrow$. Since this holds for all contexts e, we conclude that $e_1 \leq_{id_w} e_2 : ty$ does indeed imply that $e_1 \leq_{\mathrm{ctx}} e_2 : ty$. $\square$

Open Problems. The definition of $\leq_r$, with its interplay between expression-relations and frame stack-relations, was introduced to get round the difficulty of establishing the necessary fundamental properties of the logical relation (and hence property (III) of Theorem 5.2) in the presence of recursively defined functions. Note that these difficulties have to be tackled even if the particular examples of contextual equivalence we are interested in do not involve such functions (as in fact was the case in this paper). This reflects the unfortunate non-local aspect of the definition of contextual equivalence: even if the expressions we are interested in do not involve a particular language construct, we have to consider their behaviour in all contexts and the context may make use of the construct. Thus adding recursive functions complicates reasoning about non-recursive functions with local state. What other features of ML might cause trouble? I have listed some important ones in Fig. 6. There is some reason to think we could reason about the contextual equivalence of ML *structures* and *functors* using the logical relations methods outlined here: see the results about existential types in [13]. The other features—recursive mutable data, references to values of arbitrary type, and object-oriented features—are more problematic. One difficulty is that the definition of the logical relation (Definition 5.4) proceeds by induction on the structure of types. In the presence of recursive types one has to use some other approach in order to avoid a circular definition; here syntactic versions of the construction of recursively defined domains [4] may be of assistance. A more subtle problem is that some of our definitions (for example the notion of extension of state-relations in Definition 5.1) exploit the fact that we restricted attention to memory states with a very simple, 'flat' structure; many of the features listed in Fig. 6 cause memory states to have a complicated, recursive structure that blocks the use of some of the definitions as they stand.

Can the method of proving contextual equivalences outlined here be extended to larger fragments of ML with:

- structures and signatures (abstract data types)
- functions with local references to values of arbitrary types (and ditto for exception packets)
- recursively defined, mutable data structures
- OCaml-style objects and classes?

Are there other forms of logical relation, useful for proving contextual equivalences?

Fig. 6. Some things we do not yet know how to do

Finally, it should be pointed out that the simulation property of the logical relation in Theorem 5.2(I) is only a sufficient, but not a necessary condition for $e_1 \leq_{\text{ctx}} e_2 : ty$ to hold. For example

$$\begin{aligned} awk \triangleq\ & \texttt{let}\ a\ \texttt{= ref 0 in} \\ & \texttt{fun(}f\ \texttt{: unit -> unit) -> (}a\ \texttt{:= 1 ;}\ f\ \texttt{() ; !}a\texttt{)} \end{aligned} \tag{23}$$

satisfies $awk =_{\text{ctx}} (\texttt{fun(}g\ \texttt{: unit -> unit) ->}\ g\ \texttt{() ; 1}) : \texttt{(unit -> unit) -> int}$, but it is not possible to use Theorem 5.2 to prove it; see Example 5.9 of [15], which discusses this example.

6 Conclusion

We have described a method for proving contextual equivalence of ML functions involving local state, based on a certain kind of logical relation parameterised by state-relations. Theorem 5.2 summarises the properties of this logical relation that are needed for applications. However, the construction of a suitable logical relation is complicated by the presence of recursive definitions in ML. We got around this complication by using a reformulation of the structural operational semantics of ML in terms of frame stacks. This reformulation provides a structurally inductive characterisation of termination of ML evaluation that is not only used in the definition of the logical relation, but also provides a very useful tool for proving general properties of evaluation, like the Unwinding Theorem 5.3.

A Appendix: A Fragment of ML

A.1 Types

$ty ::=$	`bool`	booleans
	`int`	integers
	`unit`	unit
	`int ref`	integer storage locations
	ty `*` ty	pairs
	ty `->` ty	functions

A.2 Expressions

$e ::= x, f$	value identifiers ($x, f \in \mathit{Var}$)
`true`	boolean constants
`false`	
`if` e `then` e `else` e	conditional
`n`	integer constants ($\mathtt{n} \in \mathbb{Z}$)
e `=` e	integer equality
e `+` e	addition
e `-` e	subtraction
`()`	unit value
`!`e	look-up
e `:=` e	assignment
`ref` e	storage creation
e `==` e	location equality
e `;` e	sequence
e `,` e	pair
`fst` e	first projection
`snd` e	second projection
`fun(`x `:` ty`) ->` e	function abstraction
`fun` f `= (`x `:` ty`) ->` e	recursively defined function
$e\,e$	function application
`let` x `=` e `in` e	local definition
ℓ	storage locations ($\ell \in \mathit{Loc}$)

Notes.

1. The concrete syntax of expressions is like that of Caml rather than Standard ML (not that there are any very great differences between the two languages for the fragment we are using).
2. As well as having a canonical form for function abstractions, it simplifies the presentation of the operational semantics to have a separate canonical form `fun` f `= (`x `:` ty`) ->` e for recursively defined functions. In Caml this could be written as

   `let rec` f `= (fun(`x `:` ty`) ->` e`) in` f.

3. *Var* and *Loc* are some fixed, countably infinite sets (disjoint from each other, and disjoint from the set of integers, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$).
4. What we call *storage locations* are called *addresses* in [9]. They do not occur explicitly in the ML expressions written by users, but rather, occur implicitly via environments binding value identifiers to addresses (and to other kinds of semantic value). We will use a formulation of ML's operational semantics that does without environments, at the minor expense of having to consider an extended set of expressions, in which names of storage locations can occur explicitly.

5. We write $loc(e)$ for the finite subset of Loc consisting of all storage locations occurring in the expression e.
6. We write $fv(e)$ for the finite subset of Var consisting of all value identifiers occurring freely in the expression e. This finite set is defined by induction on the structure of e. The only interesting clauses are for the syntax-forming operations that are binders:

$$\begin{aligned} fv(\texttt{fun}(\boldsymbol{x} : ty) \texttt{ -> } e) &\triangleq fv(e) - \{\boldsymbol{x}\} \\ fv(\texttt{fun } \boldsymbol{f} \texttt{ = } (\boldsymbol{x} : ty) \texttt{ -> } e) &\triangleq fv(e) - \{\boldsymbol{f}, \boldsymbol{x}\} \\ fv(\texttt{let } \boldsymbol{x} \texttt{ = } e_1 \texttt{ in } e_2) &\triangleq fv(e_1) \cup (fv(e_2) - \{\boldsymbol{x}\}). \end{aligned}$$

A.3 Evaluation Relation

This is of the form

$$s, e \Rightarrow v, s'$$

where

– e is a *closed* expression (i.e. $fv(e)$ is empty)
– v is a closed *canonical form*, which by definition is a closed expression in the subset of expressions generated by the grammar

$$\begin{aligned} v ::= {} & \boldsymbol{x}, \boldsymbol{f} \\ & \texttt{true} \\ & \texttt{false} \\ & \texttt{n} \\ & \texttt{()} \\ & v \texttt{ , } v \\ & \texttt{fun}(\boldsymbol{x} : ty) \texttt{ -> } e \\ & \texttt{fun } \boldsymbol{f} \texttt{ = } (\boldsymbol{x} : ty) \texttt{ -> } e \\ & \ell \end{aligned}$$

– s, s' are *states*, which by definition are finite functions from Loc to $\mathbb{Z}$
– $loc(e) \subseteq dom(s)$ and $loc(v) \subseteq dom(s')$.

The evaluation relation is inductively defined by the following rules. (The notation $e[e_1/\boldsymbol{x}]$ used in some of the rules indicates the substitution of e_1 for all free occurrences of $\boldsymbol{x}$ in e; similarly, $e[e_1/\boldsymbol{x}_1, e_2/\boldsymbol{x}_2, \ldots]$ indicates simultaneous substitution; in this paper we will only need to consider the substitution of *closed* expressions, so I omit a discussion of avoiding capture of free identifiers by binders during substitution. The notation $s[\ell \mapsto \texttt{n}]$ used in some of the rules denotes the state mapping ℓ to $\texttt{n}$ and otherwise acting like s.)

Canonical forms:

$$\frac{v \text{ in canonical form}}{s, v \Rightarrow v, s}$$

Conditional:

$$\frac{s, e \Rightarrow \texttt{true}, s' \quad s', e_1 \Rightarrow v, s''}{s, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow v, s''} \qquad \frac{s, e \Rightarrow \texttt{false}, s' \quad s', e_2 \Rightarrow v, s''}{s, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow v, s''}$$

Integer equality:

$$\frac{s, e_1 \Rightarrow \texttt{n}, s' \quad s', e_2 \Rightarrow \texttt{n}, s''}{s, e_1 \texttt{=} e_2 \Rightarrow \texttt{true}, s''} \qquad \frac{s, e_1 \Rightarrow \texttt{n}_1, s' \quad s', e_2 \Rightarrow \texttt{n}_2, s'' \quad \texttt{n}_1 \neq \texttt{n}_2}{s, e_1 \texttt{=} e_2 \Rightarrow \texttt{false}, s''}$$

Arithmetic:

$$\frac{s, e_1 \Rightarrow \texttt{n}_1, s' \quad s', e_2 \Rightarrow \texttt{n}_2, s'' \quad \texttt{op} \in \{\texttt{+}, \texttt{-}\} \quad \texttt{n} \text{ is the result of combining } \texttt{n}_1 \text{ and } \texttt{n}_2 \text{ according to } \texttt{op}}{s, e_1\ \texttt{op}\ e_2 \Rightarrow \texttt{n}, s''}$$

Look-up:

$$\frac{s, e \Rightarrow \ell, s' \quad (\ell \mapsto \texttt{n}) \in s'}{s, \texttt{!}e \Rightarrow \texttt{n}, s'}$$

Assignment:

$$\frac{s, e_1 \Rightarrow \ell, s' \quad s', e_2 \Rightarrow \texttt{n}, s''}{s, e_1 \texttt{:=} e_2 \Rightarrow \texttt{()}, s''[\ell \mapsto \texttt{n}]}$$

Storage creation:

$$\frac{s, e \Rightarrow \texttt{n}, s' \quad \ell \notin \mathit{dom}(s')}{s, \texttt{ref}\ e \Rightarrow \ell, s'[\ell \mapsto \texttt{n}]}$$

Location equality:

$$\frac{s, e_1 \Rightarrow \ell, s' \quad s', e_2 \Rightarrow \ell, s''}{s, e_1 \texttt{==} e_2 \Rightarrow \texttt{true}, s''} \qquad \frac{s, e_1 \Rightarrow \ell_1, s' \quad s', e_2 \Rightarrow \ell_2, s'' \quad \ell_1 \neq \ell_2}{s, e_1 \texttt{==} e_2 \Rightarrow \texttt{false}, s''}$$

Sequence:

$$\frac{s, e_1 \Rightarrow v_1, s' \quad s', e_2 \Rightarrow v_2, s''}{s, e_1 \texttt{ ; } e_2 \Rightarrow v_2, s''}$$

Pair:

$$\frac{s, e_1 \Rightarrow v_1, s' \quad s', e_2 \Rightarrow v_2, s''}{s, (e_1 \texttt{ , } e_2) \Rightarrow (v_1 \texttt{ , } v_2), s''}$$

Projections:

$$\frac{s, e \Rightarrow (v_1 \texttt{ , } v_2), s'}{s, \texttt{fst } e \Rightarrow v_1, s'} \qquad \frac{s, e \Rightarrow (v_1 \texttt{ , } v_2), s'}{s, \texttt{snd } e \Rightarrow v_2, s'}$$

Function application:

$$\frac{s, e_1 \Rightarrow v_1, s' \quad s', e_2 \Rightarrow v_2, s'' \quad v_1 = \texttt{fun(}\boldsymbol{x} \texttt{ : } ty\texttt{) -> } e \quad s'', e[v_2/\boldsymbol{x}] \Rightarrow v_3, s'''}{s, e_1\, e_2 \Rightarrow v_3, s'''} \qquad \frac{s, e_1 \Rightarrow v_1, s' \quad s', e_2 \Rightarrow v_2, s'' \quad v_1 = \texttt{fun } \boldsymbol{f} \texttt{ = (}\boldsymbol{x} \texttt{ : } ty\texttt{) -> } e \quad s'', e[v_1/\boldsymbol{f}, v_2/\boldsymbol{x}] \Rightarrow v_3, s'''}{s, e_1\, e_2 \Rightarrow v_3, s'''}$$

Local definition:

$$\frac{s, e_1 \Rightarrow v_1, s' \quad s', e_2[v_1/\boldsymbol{x}] \Rightarrow v_2, s''}{s, \texttt{let } \boldsymbol{x} \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow v_2, s''}$$

A.4 Type Assignment Relation

This is of the form

$$\Gamma \vdash e : ty$$

where

- the *typing context* Γ is a function from a finite set $dom(\Gamma)$ of variables to types
- e is an expression
- ty is a type.

It is inductively generated by the following rules. (The notation $\Gamma[\boldsymbol{x} \mapsto ty]$ used in some of the rules indicates the typing context mapping $\boldsymbol{x}$ to ty and otherwise acting like Γ.)

Value identifiers:

$$\frac{x \in dom(\Gamma) \quad \Gamma(x) = ty}{\Gamma \vdash x : ty}$$

Boolean constants:

$$\frac{\mathtt{b} \in \{\mathtt{true}, \mathtt{false}\}}{\Gamma \vdash \mathtt{b} : \mathtt{bool}}$$

Conditional:

$$\frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash e_1 : ty \quad \Gamma \vdash e_2 : ty}{\Gamma \vdash (\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) : ty}$$

Integer constants:

$$\frac{\mathtt{n} \in \mathbb{Z}}{\Gamma \vdash \mathtt{n} : \mathtt{int}}$$

Integer equality:

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash (e_1 \mathbin{\mathtt{=}} e_2) : \mathtt{bool}}$$

Arithmetic:

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \quad \Gamma \vdash e_2 : \mathtt{int} \quad \mathtt{op} \in \{\mathtt{+}, \mathtt{-}\}}{\Gamma \vdash (e_1\ \mathtt{op}\ e_2) : \mathtt{int}}$$

Unit value:

$$\frac{}{\Gamma \vdash \mathtt{()} : \mathtt{unit}}$$

Look-up:

$$\frac{\Gamma \vdash e : \mathtt{int\,ref}}{\Gamma \vdash \mathtt{!}e : \mathtt{int}}$$

Assignment:

$$\frac{\Gamma \vdash e_1 : \mathtt{int\,ref} \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash (e_1 \mathbin{\mathtt{:=}} e_2) : \mathtt{unit}}$$

Storage creation:

$$\frac{\Gamma \vdash e : \mathtt{int}}{\Gamma \vdash \mathtt{ref}\ e : \mathtt{int\,ref}}$$

Location equality:

$$\frac{\Gamma \vdash e_1 : \mathtt{int\,ref} \quad \Gamma \vdash e_2 : \mathtt{int\,ref}}{\Gamma \vdash (e_1 \mathbin{\mathtt{==}} e_2) : \mathtt{bool}}$$

Sequence:

$$\frac{\Gamma \vdash e_1 : ty_1 \quad \Gamma \vdash e_2 : ty_2}{\Gamma \vdash (e_1\ \mathtt{;}\ e_2) : ty_2}$$

Pair:

$$\frac{\Gamma \vdash e_1 : ty_1 \quad \Gamma \vdash e_2 : ty_2}{\Gamma \vdash e_1\ \mathtt{,}\ e_2 : ty_1\ \mathtt{*}\ ty_2}$$

Projections:

$$\frac{\Gamma \vdash e : ty_1\ \mathtt{*}\ ty_2}{\Gamma \vdash \mathtt{fst}\ e : ty_1} \qquad \frac{\Gamma \vdash e : ty_1\ \mathtt{*}\ ty_2}{\Gamma \vdash \mathtt{snd}\ e : ty_2}$$

Function abstraction:

$$\frac{\Gamma[\boldsymbol{x} \mapsto ty_1] \vdash e : ty_2 \quad \boldsymbol{x} \notin dom(\Gamma}{\Gamma \vdash (\mathtt{fun(}\boldsymbol{x}\ \mathtt{:}\ ty_1\mathtt{)}\ \mathtt{->}\ e) : ty_1\ \mathtt{->}\ ty_2}$$

Recursively defined function:

$$\frac{\Gamma[\boldsymbol{f} \mapsto ty_1\ \mathtt{->}\ ty_2][\boldsymbol{x} \mapsto ty_1] \vdash e : ty_2 \quad \boldsymbol{f}, \boldsymbol{x} \notin dom(\Gamma) \quad \boldsymbol{f} \neq \boldsymbol{x}}{\Gamma \vdash (\mathtt{fun}\ \boldsymbol{f}\ \mathtt{=}\ \mathtt{(}\boldsymbol{x}\ \mathtt{:}\ ty_1\mathtt{)}\ \mathtt{->}\ e) : ty_1\ \mathtt{->}\ ty_2}$$

Function application:

$$\frac{\Gamma \vdash e_1 : ty_2\ \mathtt{->}\ ty_1 \quad \Gamma \vdash e_2 : ty_2}{\Gamma \vdash e_1\, e_2 : ty_1}$$

Local definition:

$$\frac{\Gamma \vdash e_1 : ty_1 \qquad \Gamma[\boldsymbol{x} \mapsto ty_1] \vdash e_2 : ty_2 \qquad \boldsymbol{x} \notin dom(\Gamma)}{\Gamma \vdash (\texttt{let } \boldsymbol{x} \texttt{ = } e_1 \texttt{ in } e_2) : ty_2}$$

Storage locations:

$$\frac{\ell \in Loc}{\Gamma \vdash \ell : \texttt{int ref}}$$

Theorem A.1 (Type soundness).

$$(e, s \Rightarrow v, s') \; \& \; (\emptyset \vdash e : ty) \;\supset\; (\emptyset \vdash v : ty).$$ □

A.5 Transition Relation

This is of the form

$$(s\ ,\ e) \to (s'\ ,\ e')$$

where e, e' are closed expressions and s, s' are memory states with $loc(e) \subseteq dom(s)$ and $loc(e') \subseteq dom(s')$. The transition relation is inductively defined by the following rules.

Basic reductions:

$$\frac{}{(s\ ,\ \texttt{if true then } e_1 \texttt{ else } e_2) \to (s\ ,\ e_1)}$$

$$\frac{}{(s\ ,\ \texttt{if false then } e_1 \texttt{ else } e_2) \to (s\ ,\ e_2)}$$

$$\frac{}{(s\ ,\ \texttt{n} = \texttt{n}) \to (s\ ,\ \texttt{true})} \qquad \frac{\texttt{n}_1 \neq \texttt{n}_2}{(s\ ,\ \texttt{n}_1 = \texttt{n}_2) \to (s\ ,\ \texttt{false})}$$

$$\frac{\texttt{op} \in \{\texttt{+}, \texttt{-}\} \qquad \texttt{n} \text{ is the result of combining } \texttt{n}_1 \text{ and } \texttt{n}_2 \text{ according to } \texttt{op}}{(s\ ,\ \texttt{n}_1 \texttt{ op } \texttt{n}_2) \to (s\ ,\ \texttt{n})}$$

$$\frac{(\ell \mapsto \texttt{n}) \in s}{(s\ ,\ \texttt{!}\ell) \to (s\ ,\ \texttt{n})}$$

$$\frac{}{(s\ ,\ \ell \texttt{ := } \texttt{n}) \to (s[\ell \mapsto \texttt{n}]\ ,\ \texttt{()})}$$

$$\frac{\ell \notin dom(s)}{(s\ ,\ \texttt{ref n}) \to (s[\ell \mapsto \texttt{n}]\ ,\ \ell)}$$

$$\frac{}{(s\ ,\ \ell\ \texttt{==}\ \ell) \to (s\ ,\ \texttt{true})} \qquad \frac{\ell_1 \neq \ell_2}{(s\ ,\ \ell_1\ \texttt{==}\ \ell_2) \to (s\ ,\ \texttt{false})}$$

$$\frac{v \text{ a canonical form}}{(s\ ,\ (v\ \texttt{;}\ e)) \to (s\ ,\ e)}$$

$$\frac{v_1, v_2 \text{ canonical forms}}{(s\ ,\ \texttt{fst}\ (v_1\ \texttt{,}\ v_2)) \to (s\ ,\ v_1)} \qquad \frac{v_1, v_2 \text{ canonical forms}}{(s\ ,\ \texttt{snd}\ (v_1\ \texttt{,}\ v_2)) \to (s\ ,\ v_1)}$$

$$\frac{\begin{array}{l} v_1 = \texttt{fun(}\boldsymbol{x}\ \texttt{:}\ ty\texttt{) ->}\ e \\ v_2 \text{ a canonical form} \end{array}}{(s\ ,\ v_1\, v_2) \to (s\ ,\ e[v_2/\boldsymbol{x}])} \qquad \frac{\begin{array}{l} v_1 = \texttt{fun}\ \boldsymbol{f}\ \texttt{= (}\boldsymbol{x}\ \texttt{:}\ ty\texttt{) ->}\ e \\ v_2 \text{ a canonical form} \end{array}}{(s\ ,\ v_1\, v_2) \to (s\ ,\ e[v_1/\boldsymbol{f}, v_2/\boldsymbol{x}])}$$

$$\frac{v \text{ a canonical form}}{(s\ ,\ \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ v\ \texttt{in}\ e) \to (s\ ,\ e[v/\boldsymbol{x}])}$$

Simplification steps:

$$\frac{(s\ ,\ e) \to (s'\ ,\ e')}{(s\ ,\ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \to (s'\ ,\ \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2)}$$

$$\frac{\begin{array}{l} (s\ ,\ e_1) \to (s'\ ,\ e_1') \\ \texttt{op} \in \{\texttt{=}, \texttt{+}, \texttt{-}, \texttt{:=}, \texttt{==}, \texttt{;}, \texttt{,}\} \end{array}}{(s\ ,\ e_1\ \texttt{op}\ e_2) \to (s'\ ,\ e_1'\ \texttt{op}\ e_2)} \qquad \frac{\begin{array}{l} (s\ ,\ e) \to (s'\ ,\ e') \\ v \text{ a canonical form} \\ \texttt{op} \in \{\texttt{=}, \texttt{+}, \texttt{-}, \texttt{:=}, \texttt{==}, \texttt{,}\} \end{array}}{(s\ ,\ v\ \texttt{op}\ e) \to (s'\ ,\ v\ \texttt{op}\ e')}$$

$$\frac{\begin{array}{l} (s\ ,\ e_1) \to (s'\ ,\ e_1') \\ \texttt{op} \in \{\texttt{!}, \texttt{ref}\ , \texttt{fst}\ , \texttt{snd}\ \} \end{array}}{(s\ ,\ \texttt{op}\, e) \to (s'\ ,\ \texttt{op}\, e')}$$

$$\frac{(s\ ,\ e_1) \to (s'\ ,\ e_1')}{(s\ ,\ e_1\, e_2) \to (s'\ ,\ e_1'\, e_2)} \qquad \frac{\begin{array}{l} (s\ ,\ e) \to (s'\ ,\ e') \\ v \text{ a canonical form} \end{array}}{(s\ ,\ v\, e) \to (s'\ ,\ v\, e')}$$

$$\frac{(s\ ,\ e_1) \to (s'\ ,\ e_1')}{(s\ ,\ \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ e_1\ \texttt{in}\ e_2) \to (s'\ ,\ \texttt{let}\ \boldsymbol{x}\ \texttt{=}\ e_1'\ \texttt{in}\ e_2)}$$

Theorem A.2 (The relationship between evaluation and transition).

$$(s, e \Rightarrow v, s') \quad \equiv \quad (s\ ,\ e) \to^* (s'\ ,\ v)$$

where $\to^*$ *is the reflexive-transitive closure of* $\to$. □

A.6 An Abstract Machine

The configurations of the machine take the form $\langle s\ ,\ \mathcal{F}s\ ,\ e\rangle$ where s is a state (cf. Sect. A.3), e is a closed expression (cf. Sect. A.2) and $\mathcal{F}s$ is a closed frame stack. The *frame stacks* are given by:

$$\begin{array}{rll} \mathcal{F}s ::= & \mathcal{I}d & \text{empty} \\ & \mathcal{F}s \circ \mathcal{F} & \text{non-empty} \end{array}$$

where $\mathcal{F}$ is an *evaluation frame*:

$$\begin{array}{rll} \mathcal{F} ::= & \mathtt{if}\ [-]\ \mathtt{then}\ e\ \mathtt{else}\ e & \\ & [-]\ \mathtt{op}\ e & \text{for } \mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{;}, \mathtt{,}\} \\ & v\ \mathtt{op}\ [-] & \text{for } \mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{,}\} \\ & \mathtt{op}\ [-] & \text{for } \mathtt{op} \in \{\mathtt{!}, \mathtt{ref}\ , \mathtt{fst}\ , \mathtt{snd}\ \} \\ & [-]\ e & \\ & v\ [-] & \\ & \mathtt{let}\ \boldsymbol{x}\ \mathtt{=}\ [-]\ \mathtt{in}\ e & \end{array}$$

(where e ranges over expressions and v over expressions in canonical form). The set $fv(\mathcal{F}s)$ of free value identifiers of a frame stack $\mathcal{F}s$ are all those value identifiers occurring freely in its constituent expressions; $\mathcal{F}s$ is *closed* if $fv(\mathcal{F}s)$ is empty.

The transitions of the abstract machine, $\langle s\ ,\ \mathcal{F}s\ ,\ e\rangle \to \langle s'\ ,\ \mathcal{F}s'\ ,\ e'\rangle$, are defined by case analysis of, firstly, the structure of e and then the structure of $\mathcal{F}s$:

Case $e = v$ is in canonical form:

$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{if}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ e_1\rangle$, if $v = \mathtt{true}$
$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{if}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ e_2\rangle$, if $v = \mathtt{false}$
$\langle s\ ,\ \mathcal{F}s \circ ([-]\ \mathtt{op}\ e)\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s \circ (v\ \mathtt{op}\ [-])\ ,\ e\rangle$, for $\mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{,}\}$
$\langle s\ ,\ \mathcal{F}s \circ (v'\ \mathtt{op}\ [-])\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ v''\rangle$,
if v'' is the result of combining v' and v according to $\mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{==}, \mathtt{,}\}$
$\langle s\ ,\ \mathcal{F}s \circ (\ell\ \mathtt{:=}\ [-])\ ,\ v\rangle \to \langle s[\ell \mapsto \mathtt{n}]\ ,\ \mathcal{F}s\ ,\ \mathtt{()}\rangle$, if $v = \mathtt{n}$
$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{!}[-])\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{n}\rangle$, if $v = \ell$ and $(\ell \mapsto \mathtt{n}) \in dom(s)$
$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{ref}\ [-])\ ,\ v\rangle \to \langle s[\ell \mapsto \mathtt{n}]\ ,\ \mathcal{F}s\ ,\ \ell\rangle$, if $v = \mathtt{n}$ and $\ell \notin dom(s)$
$\langle s\ ,\ \mathcal{F}s \circ ([-]\ \mathtt{;}\ e)\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ e\rangle$
$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{fst}\ [-])\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ v_1\rangle$, if $v = (v_1\ \mathtt{,}\ v_2)$
$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{snd}\ [-])\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ v_2\rangle$, if $v = (v_1\ \mathtt{,}\ v_2)$
$\langle s\ ,\ \mathcal{F}s \circ ([-]\ e)\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s \circ (v\ [-])\ ,\ e\rangle$
$\langle s\ ,\ \mathcal{F}s \circ (v'\ [-])\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ e[v/\boldsymbol{x}]\rangle$, if $v' = \mathtt{fun(}\boldsymbol{x}\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e$
$\langle s\ ,\ \mathcal{F}s \circ (v'\ [-])\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ e[v'/\boldsymbol{f}, v/\boldsymbol{x}]\rangle$, if $v' = \mathtt{fun}\ \boldsymbol{f}\ \mathtt{=}\ \mathtt{(}\boldsymbol{x}\ \mathtt{:}\ ty\mathtt{)}\ \mathtt{->}\ e$
$\langle s\ ,\ \mathcal{F}s \circ (\mathtt{let}\ \boldsymbol{x}\ \mathtt{=}\ [-]\ \mathtt{in}\ e)\ ,\ v\rangle \to \langle s\ ,\ \mathcal{F}s\ ,\ e[v/\boldsymbol{x}]\rangle$

Case e is not in canonical form:

$\langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \to \langle s\ ,\ \mathcal{F}s \circ (\mathtt{if}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)\ ,\ e\rangle$
$\langle s\ ,\ \mathcal{F}s\ ,\ e_1\ \mathtt{op}\ e_2\rangle \to \langle s\ ,\ \mathcal{F}s \circ ([-]\ \mathtt{op}\ e_2)\ ,\ e_1\rangle$, for $\mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{;}., \mathtt{,}\}$
$\langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{op}\ e\rangle \to \langle s\ ,\ \mathcal{F}s \circ (\mathtt{op}\ [-])\ ,\ e\rangle$, for $\mathtt{op} \in \{\mathtt{!}, \mathtt{ref}\ , \mathtt{fst}\ , \mathtt{snd}\ \}$
$\langle s\ ,\ \mathcal{F}s\ ,\ e_1\ e_2\rangle \to \langle s\ ,\ \mathcal{F}s \circ ([-]\ e_2)\ ,\ e_1\rangle$
$\langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{let}\ \boldsymbol{x}\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2\rangle \to \langle s\ ,\ \mathcal{F}s \circ (\mathtt{let}\ \boldsymbol{x}\ \mathtt{=}\ [-]\ \mathtt{in}\ e_2)\ ,\ e_1\rangle$

Theorem A.3 (The relationship between evaluation and the abstract machine).

$$\langle s \, , \mathcal{F}s \, , e\rangle \to^* \langle s' \, , \mathcal{I}d \, , v\rangle \quad \equiv \quad (s, \mathcal{F}s[e] \Rightarrow v, s')$$

where the application $\mathcal{F}s[e]$ of a frame stack $\mathcal{F}s$ to an expression e is defined by induction on the length of $\mathcal{F}s$ as follows:

$$\begin{cases} \mathcal{I}d[e] \triangleq e \\ (\mathcal{F}s \circ \mathcal{F})[e] \triangleq \mathcal{F}s[\mathcal{F}[e]] \end{cases}$$

(each evaluation frame $\mathcal{F}$ is an evaluation context containing a hole $[-]$ that can be replaced by e to obtain an expression $\mathcal{F}[e]$). □

A.7 A Structurally Inductive Termination Relation

This is of the form

$$\langle s \, , \mathcal{F}s \, , e\rangle\searrow$$

where s is a memory state, $\mathcal{F}s$ a frame stack and e a closed expression. It is inductively defined by the following rules.

$$\frac{v \text{ a canonical form}}{\langle s \, , \mathcal{I}d \, , v\rangle\searrow}$$

$$\frac{\langle s \, , \mathcal{F}s \, , e_1\rangle\searrow \qquad v = \mathtt{true}}{\langle s \, , \mathcal{F}s \circ (\mathtt{if}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \, , v\rangle\searrow}$$

$$\frac{\langle s \, , \mathcal{F}s \, , e_2\rangle\searrow \qquad v = \mathtt{false}}{\langle s \, , \mathcal{F}s \circ (\mathtt{if}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \, , v\rangle\searrow}$$

$$\frac{\langle s \, , \mathcal{F}s \circ (v\ \mathtt{op}\ [-]) \, , e\rangle\searrow \qquad \mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{,}\}}{\langle s \, , \mathcal{F}s \circ ([-]\ \mathtt{op}\ e) \, , v\rangle\searrow}$$

$$\frac{\langle s \, , \mathcal{F}s \, , v''\rangle\searrow \qquad \mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{,}\} \qquad v'' \text{ is result of combining } v' \text{ and } v \text{ according to } \mathtt{op}}{\langle s \, , \mathcal{F}s \circ (v'\ \mathtt{op}\ [-]) \, , v\rangle\searrow}$$

$$\frac{\langle s[\ell \mapsto \mathtt{n}] \, , \mathcal{F}s \, , \mathtt{()}\rangle\searrow \qquad v = \mathtt{n}}{\langle s \, , \mathcal{F}s \circ (\ell\ \mathtt{:=}\ [-]) \, , v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{n}\rangle\searrow \quad v = \ell \quad (\ell \mapsto \mathtt{n}) \in \mathit{dom}(s)}{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{!}\,[-])\ ,\ v\rangle\searrow}$$

$$\frac{\langle s[\ell \mapsto \mathtt{n}]\ ,\ \mathcal{F}s\ ,\ \ell\rangle\searrow \quad v = \mathtt{n} \quad \ell \notin \mathit{dom}(s)}{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{ref}\ [-])\ ,\ v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s\ ,\ e\rangle\searrow}{\langle s\ ,\ \mathcal{F}s \circ ([-]\ \mathtt{;}\ e)\ ,\ v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s\ ,\ v_1\rangle\searrow \quad v = (v_1\ \mathtt{,}\ v_2)}{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{fst}\ [-])\ ,\ v\rangle\searrow} \qquad \frac{\langle s\ ,\ \mathcal{F}s\ ,\ v_2\rangle\searrow \quad v = (v_1\ \mathtt{,}\ v_2)}{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{snd}\ [-])\ ,\ v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s \circ (v\,[-])\ ,\ e\rangle\searrow}{\langle s\ ,\ \mathcal{F}s \circ ([-]\,e)\ ,\ v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s\ ,\ e[v/\boldsymbol{x}]\rangle\searrow \quad v' = \mathtt{fun(}\boldsymbol{x}\ \mathtt{:}\ \mathit{ty}\mathtt{)\ ->}\ e}{\langle s\ ,\ \mathcal{F}s \circ (v'\,[-])\ ,\ v\rangle\searrow} \qquad \frac{\langle s\ ,\ \mathcal{F}s\ ,\ e[v'/\boldsymbol{f}, v/\boldsymbol{x}]\rangle\searrow \quad v' = \mathtt{fun}\ \boldsymbol{f}\ \mathtt{=\ (}\boldsymbol{x}\ \mathtt{:}\ \mathit{ty}\mathtt{)\ ->}\ e}{\langle s\ ,\ \mathcal{F}s \circ (v'\,[-])\ ,\ v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s\ ,\ e[v/\boldsymbol{x}]\rangle\searrow}{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{let}\ \boldsymbol{x}\ \mathtt{=}\ [-]\ \mathtt{in}\ e)\ ,\ v\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{if}\ [-]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)\ ,\ e\rangle\searrow}{\langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s \circ ([-]\ \mathtt{op}\, e_2)\ ,\ e_1\rangle\searrow \quad \mathtt{op} \in \{\mathtt{=}, \mathtt{+}, \mathtt{-}, \mathtt{:=}, \mathtt{==}, \mathtt{,}\}}{\langle s\ ,\ \mathcal{F}s\ ,\ e_1\ \mathtt{op}\, e_2\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s \circ (\mathtt{op}\,[-])\ ,\ e\rangle\searrow \quad \mathtt{op} \in \{\mathtt{!}, \mathtt{ref}\ , \mathtt{fst}\ , \mathtt{snd}\ \}}{\langle s\ ,\ \mathcal{F}s\ ,\ \mathtt{op}\, e\rangle\searrow}$$

$$\frac{\langle s\ ,\ \mathcal{F}s \circ ([-]\,e_2)\ ,\ e_1\rangle\searrow}{\langle s\ ,\ \mathcal{F}s\ ,\ e_1\, e_2\rangle\searrow}$$

Comparing the abstract machine steps in Sect. A.6 with the above rules it is not hard to see that we have:

Theorem A.4.

$$\langle s\ ,\mathcal{F}s\ ,e\rangle\searrow \quad \equiv \quad \exists s', v\ \ (\langle s\ ,\mathcal{F}s\ ,e\rangle \to^* \langle s'\ ,\mathcal{I}d\ ,v\rangle)\ \ .$$

Hence by Theorem A.3, $s, e \Downarrow$ *holds if and only if* $\langle s\ ,\mathcal{I}d\ ,e\rangle\searrow$ *does.* □

B Exercises

Exercise B.1. Let f, g and t be defined as in equations (3), (4) and (5). Use the rules in Sect. A.3 to prove that

$$\emptyset, t\,f \Rightarrow \mathtt{false}, s \quad \text{and} \quad \emptyset, t\,g \Rightarrow \mathtt{true}, s$$

hold for some state s. (Here $\emptyset$ denotes the *empty state*, whose domain of definition is $dom(\emptyset) = \emptyset$, the empty set of locations.)

Exercise B.2. Prove the type soundness property of evaluation stated in Theorem A.1. Use induction on the derivation of the evaluation $e, s \Rightarrow v, s'$. You will first need to prove the following substitution property of the type assignment relation:

$$\Gamma \vdash e : ty \ \ \&\ \ \Gamma[\boldsymbol{x} \mapsto ty] \vdash e' : ty' \ \supset\ \Gamma \vdash e'[e/\boldsymbol{x}] : ty'.$$

Exercise B.3. Given $e_1, e_2 \in \mathrm{Prog}_{ty}$, define $e_1 \leq_{obs} e_2 : ty$ to mean that for all $\boldsymbol{x} : ty \vdash e : ty'$ and all states s

$$s, e[e_1/\boldsymbol{x}] \Rightarrow v_1, s_1 \ \supset\ \exists v_2, s_2\,.\,(s, e[e_2/\boldsymbol{x}] \Rightarrow v_2, s_2)\ \ \&\ \ obs(v_1, s_1) = obs(v_2, s_2)$$

where the function obs is defined in equation (2). Prove that $e_1 \leq_{obs} e_2 : ty$ holds if and only if $e_1 \leq_{\mathrm{ctx}} e_2 : ty$ does.

Exercise B.4. Prove Theorem A.2 relating the evaluation and transition relations of ML. First prove

$$(s\ ,e) \to (s'\ ,e')\ \supset\ \forall v, s''.\,(s', e' \Rightarrow v, s'')\ \supset\ (s, e \Rightarrow v, s'')$$

by induction on the derivation of $(s\ ,e) \to (s'\ ,e')$; deduce that if $(s\ ,e) \to^* (s'\ ,v)$, then $s, e \Rightarrow v, s'$. Prove the converse by induction on the derivation of $s, e \Rightarrow v, s'$.

Exercise B.5. Given $r \in \mathrm{Rel}(w_1, w_2)$ and $r' \in \mathrm{Rel}(w_1', w_2')$ with $w_1' \supseteq w_1$ and $w_2' \supseteq w_2$, show that $r' \rhd r$ (Definition 5.1) holds if and only if for all $(s_1', s_2') \in r'$

$$(s_1'\restriction_{w_1}, s_2'\restriction_{w_2}) \in r\ \ \&\ \ \forall (s_1, s_2) \in r\,.\,(s_1's_1, s_2's_2) \in r'.$$

(Here $s\restriction_w$ denotes the restriction of the function s to w; and $s's$ is the state determined by the states s' and s as in Definition 5.1.)

Exercise B.6. Use the Unwinding Theorem 5.3 to prove the property of $\leq_{\mathrm{ctx}}$ stated in equation (9).

Exercise B.7. Suppose $f \in \mathrm{Prog}_{\texttt{int->int}}$ is a closed expression with $loc(f) = \emptyset$ and such that for all $\mathtt{n} \in \mathbb{Z}$ it is the case that $\emptyset, f\,\mathtt{n}\Downarrow$ holds. Show that $f =_{\mathrm{ctx}} \mathit{memo\_f} : \texttt{int -> int}$ where

$$
\begin{aligned}
\mathit{memo\_f} \;\triangleq\;& \texttt{let } a \texttt{ = ref 0 in}\\
& \texttt{let } r \texttt{ = ref (} f\,\texttt{0) in}\\
& \texttt{fun(} x \texttt{ : int) -> (if } x \texttt{ = !} a \texttt{ then ()}\\
& \qquad\qquad\qquad\qquad \texttt{else (} a \texttt{ := } x \texttt{ ; } r \texttt{ := } f\,x\texttt{)) ; !} r .
\end{aligned}
$$

(See [15, Example 5.7], if you get stuck.)

C List of Notation

&	logical conjunction.
$\supset$	logical implication.
$\equiv$	logical bi-implication.
$=_{\mathrm{ctx}}$	contextual equivalence—see Definition 3.1.
$\leq_{\mathrm{ctx}}$	contextual preorder—see Definition 3.1.
$\leq_r$	logical simulation relation—see Theorem 5.2 and Definition 5.4.
$\rhd$	extension relation between state-relations—see Definition 5.1.
$dom(f)$	the domain of definition of a partial function f.
$e[e_1/\boldsymbol{x}]$	expression resulting from the substitution of expression e_1 for all free occurrences of $\boldsymbol{x}$ in expression e.
$e[e_1/\boldsymbol{x}_1, \ldots, e_n/\boldsymbol{x}_n]$	expression resulting from the simultaneous substitution of expression e_i for all free occurrences of $\boldsymbol{x}_i$ in expression e (for $i = 1, \ldots, n$).
$\mathcal{E}$	an evaluation context—see equation (7) in Sect. 4.
$\mathcal{E}[e]$	the expression resulting from replacing the 'hole' $[-]$ in an evaluation context $\mathcal{E}$ by the expression e.
$\mathcal{F}$	an evaluation frame, special case of an evaluation context—see Sect. A.6.
$\mathcal{F}[e]$	the expression resulting from replacing the 'hole' $[-]$ in an evaluation frame $\mathcal{F}$ by the expression e.
$\mathcal{F}s$	a frame stack—see Sect. A.6.
$\mathcal{F}s[e]$	the expression resulting from applying the frame stack $\mathcal{F}s$ to the expression e—see Theorem A.3.
$\vdash \mathcal{F}s : ty \multimap ty'$	type assignment relation for frame stacks—see (12).
$f[x \mapsto y]$	a partial function mapping x to y and otherwise acting like the partial function$f$.

$fv(e)$	finite set of free value identifiers of an expression e.
id_w	identity state-relation at world w—see Theorem 5.2(III).
$\Gamma \vdash e : ty$	type assignment relation—see Sect. A.4.
Loc	the fixed set of names of storage locations.
$loc(e)$	finite set of storage locations occurring in the expression e.
$\emptyset$	the empty set; the empty partial function; the empty state; the empty typing context.
Prog_{ty}	the set of closed expressions of type ty—see Definition 5.1.
$\mathrm{Prog}_{ty}(w)$	the set of closed expressions of type ty with locations in the finite set w.
$\mathrm{Rel}(w_1, w_2)$	the set of binary relations between states in $\mathrm{St}(w_1)$ and in $\mathrm{St}(w_1)$.
$s, e \Rightarrow v, s'$	evaluation relation—see Sect. A.3.
$s, e \Downarrow$	termination relation derived from the evaluation relation $\Rightarrow$—see Definition 3.1.
$(s\ ,\ e) \to (s'\ ,\ e')$	transition relation—see Sect. A.5.
$\langle s\ ,\ \mathcal{F}s\ ,\ e\rangle \to \langle s'\ ,\ \mathcal{F}s'\ ,\ e'\rangle$	transition of the abstract machine—see Sect. A.6.
$\langle s\ ,\ \mathcal{F}s\ ,\ e\rangle\searrow$	structurally inductive termination relation—see Sect. A.7.
$\mathrm{St}(w)$	the set of memory states defined on a finite set w of locations (i.e. $\mathbb{Z}^w$, the set of functions from w to $\mathbb{Z}$).
$\mathrm{Stack}_{ty}(w)$	the set of closed frame stacks taking an argument of type ty and with locations in a finite set w of locations—see (12).
$\mathrm{Val}_{ty}(w)$	the set of canonical forms of type ty with locations in a finite set w of locations—see Sect. A.3.
w	a finite subset of Loc, regarded as a 'world'.
$\mathbb{Z}$	the set of integers, $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

References

1. K. Aboul-Hosn and J. Hannan. Program equivalence with private state. Preprint., January 2002.
2. P. N. Benton and A. J. Kennedy. Monads, effects and transformations. In A. D. Gordon and A. M. Pitts, editors, *HOOTS '99 Higher Order Operational Techniques in Semantics, Paris, France, September 1999*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

3. G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Fourth International Workshop on Higher Order Operational Techniques in Semantics, Montréal*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2000.
4. L. Birkedal and R. Harper. Relational interpretation of recursive types in an operational setting (Summary). In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS'97, Sendai, Japan, September 23 - 26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1997.
5. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
6. R. Harper and C. Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, Carnegie Mellon University, Pittsburgh, PA, 1997.
7. A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. In *14th Annual Symposium on Logic in Computer Science*, pages 56–66. IEEE Computer Society Press, Washington, 1999.
8. G. Kahn. Natural semantics. Rapport de Recherche 601, INRIA, Sophia-Antipolis, France, 1987.
9. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
10. P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120:107–116, 1995.
11. A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.
12. A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-Like Languages*, volume 2, chapter 17, pages 173–193. Birkhauser, 1997. First appeared in *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.
13. A. M. Pitts. Existential types: Logical relations and operational equivalence. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.
14. A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.
15. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.
16. G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, School of Artificial Intelligence, University of Edinburgh, 1973.
17. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.
18. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.