A Fully Abstract Translation between a λ-Calculus with Reference Types and Standard ML

Eike Ritter*
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK

Andrew M. Pitts†
Cambridge University Computer Laboratory
Pembroke Street
Cambridge CB2 3QG, UK

September 1994

Abstract

This paper describes a syntactic translation for a substantial fragment of the core Standard ML language into a typed λ-calculus with recursive types and imperative features in the form of reference types. The translation compiles SML's use of declarations and pattern matching into λ-terms, and transforms the use of environments in the operational semantics into a simpler relation of evaluation to canonical form. The translation is shown to be 'fully abstract', in the sense that it both preserves and reflects observational equivalence (also commonly called contextual equivalence). A co-inductively defined notion of applicative bisimilarity for lambda calculi with state is developed to establish this result.

*Research partially supported by EU ESPRIT project CLICS-II.

†Research partially supported by UK EPSRC project GR/G53279.

1 Introduction

To apply techniques from operational and denotational semantics to higher order functional programming languages it is often convenient to define a translation of the programming language into a typed λ-calculus with appropriate additional features and study the typed λ-calculus instead. The transfer of semantic results about the typed λ-calculus back to ones about the original programming language relies on properties of the translation which can be rather hard to establish and which seem to be addressed rather rarely in the literature. Here we study such a translation for a substantial fragment of the core Standard ML language [MTH90] containing recursively defined datatypes and imperative features in the form of reference types. The translation compiles SML's use of declarations and pattern matching into λ-terms, and transforms the use of environments in the operational semantics into a simpler relation of evaluation to canonical form. We show that this translation is 'fully abstract', in the sense that it both preserves and reflects observational equivalence (also commonly called contextual equivalence).

Riecke [Rie90] investigates a similar problem, namely how to show which translations between call-by-value and call-by-name PCF are fully abstract. He defines a fully abstract model for the two languages to examine the translations. He transforms the question of preserving observational equivalence into the question whether a term in one language and its translation have meanings in the fully abstract models that are related by a logical relation. This method works only if fully abstract domain-theoretic models of the languages are easily available. This is true for the simple language PCF (albeit after the addition of a parallel-or function), but definitely not for SML. Indeed the presence of dynamically created, mutable storage locations for recursively defined data involving higher order functions makes the construction of adequate, let alone fully abstract, denotational models difficult in theory and hard to use in practice.

In this paper we establish the full abstraction property of our translation using methods based on operational semantics rather than denotational semantics. This entails working directly with the notion of observational equivalence of language expressions, rather than with equality of their denotations. However, the quantification over program contexts in the definition of observational equivalence makes it rather unamenable to the standard techniques associated with operational semantics (such as various forms of structural induction). For the simple language PCF, Milner's context lemma [Mil77] overcomes this problem by showing that one can restrict to a very simple collection of applicative contexts when establishing instances of PCF observational equivalence. Inspired by Milner's result, Mason and Talcott [MT91, **ciu** Theorem] prove a much harder context lemma for an untyped λ-calculus with mutable local storage locations. Although the restriction on contexts in their result is not as great as one might hope for, nevertheless they show that a substantial number of properties of observational equivalence result from it. However, Milner's result can be usefully generalized in a different direction. It is equivalent to a characterization of observational equivalence in PCF in terms of a co-inductively defined notion of applicative bisimilarity. Such notions were transferred from the study of concurrent processes to untyped λ-calculus by Abramsky [AO93] and have been applied to various kinds of lambda calculi, type theory and pure functional programming by Howe [How89], Ong [Ong93] and Gordon [Gor93]. Here we introduce a notion of applicative bisimilarity for functional languages with state and show that Howe's method in [How89] can be adapted to prove that it is a congruence. The difficulty that has to be overcome to do this for languages with state concerns the more complicated form of the operational semantics compared with that for pure functional languages. We were guided to our definition by analogy with the co-inductive characterization of equality in recursively defined domains given in [Pit94], applied to the domains needed for a denotational semantics of the kind of typed λ-calculus considered here.

As a consequence of its congruence property, the relation of applicative bisimilarity implies that of observational equivalence. Unlike the situation for non-imperative (deterministic) functional languages, the two notions do not coincide. This is due to the subtle nature of observational equivalence in the presence of dynamically created local state: see [PS93], for example. Nevertheless the notion of applicative bisimilarity we give has sufficient properties (all relatively easy to establish, apart from the congruence property) to yield the full abstraction properties of our translation of SML into a typed λ-calculus.

The paper is structured as follows. In section 2 we define the typed λ-calculus we consider, the corresponding fragment of Standard ML, and translations between the calculi in each direction. The next section defines the notion of applicative bisimulation used to show the full abstractness of the translations, and proves that it is a congruence relation. Afterwards we apply this result to show that both translations preserve and reflect observational equivalence and are in fact mutually inverse to each other up to observational equivalence.

2 The Calculi

First we define a λ-calculus with reference types, λML, and then the appropriate fragment of Standard ML.

2.1 The λ-Calculus λML

We consider a typed λ-calculus with the following features: higher-order, recursively defined functions; (local, monomorphic) references to mutable locations; and globally defined, mutually recursive datatypes. In fact we define a family of languages, parameterized by a function

$$decl : TypConst \to Typ$$

used to give the (recursive) definition $\kappa = decl(\kappa)$ of some *type constants* κ in the finite set $TypConst$, and where Typ, the collection of *types*, is defined by the grammar:

$$\begin{array}{rcll} \sigma & ::= & \kappa & \kappa \in TypConst \\ & & \mid \mathtt{unit} & \text{one element type} \\ & & \mid \mathtt{bool} & \text{two element type} \\ & & \mid \sigma \times \sigma & \text{product type} \\ & & \mid \sigma \Rightarrow \sigma & \text{function type} \\ & & \mid \sigma\,\mathtt{ref} & \text{reference type} \end{array}$$

The type constants κ act like mutually recursively defined datatypes that are global and static in the sense that they are fixed once and for all when the language is defined. There are no construction mechanisms for building new recursive datatypes from old ones in the language itself.

The expressions of the calculus are given as follows, where x is taken from an infinite set of identifiers and a from an infinite set of address names:

$$\begin{array}{rcl} e & ::= & x \mid \mathtt{in}^{\kappa}(e) \mid \mathtt{out}(e) \mid () \mid e := e \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid e = e \\ & \mid & (e,e) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \mid \lambda x^{\sigma}.e \mid ee \mid \mathtt{rec}\ f(x) =_{\sigma} e\ \mathtt{in}\ f(e) \mid \mathtt{ref}(e) \mid {!e} \mid a \end{array}$$

The expression $\texttt{rec}\ f(x) =_\sigma e'\ \texttt{in}\ f(e)$ defines f as a recursive function and applies it to e, and $!e$ accesses the value stored in the address to which e evaluates. The notion of bound and free variables are defined as usual. An expression is *closed* if it has no free identifiers. The *canonical expressions* are the closed expressions generated by the grammar:

$$c \ ::= \ \texttt{in}^\kappa(c) \mid () \mid \texttt{true} \mid \texttt{false} \mid (c, c) \mid \lambda x^\sigma.e \mid a$$

Remark We do not have any sum types or alternate clauses in the definition of recursive datatypes. Those constructions are used in modelling standard datatypes like lists. This omission simplifies the presentation—see Section 2.3 for the reason, but the same ideas that are described in this paper yield the full abstraction of the translations for the extended calculus as well.

A *type assignment* Γ is a finite function from identifiers to types. A typing assertion takes the form

$$\Gamma \vdash e : \sigma$$

where Γ is a type assignment, e an expression, σ a type and the identifiers of e are contained in the domain of Γ. The rules are listed in the appendix. Note that for every expression there is at most one type σ such that $\Gamma \vdash e : \sigma$.

A *state* s is a finite function from addresses to canonical expressions such that all addresses contained in the canonical expressions are contained in the domain of s. We write $s + \{a \mapsto c\}$ for the state obtained from s by extending its domain by $a \notin dom(s)$ and mapping a to c. We write $s \cup \{a \mapsto c\}$ for the state obtained from s by mapping $a \in dom(s)$ to c and otherwise acting like s.

The evaluation relation takes the form

$$\langle s, e \rangle \Downarrow \langle s', c \rangle$$

where s and s' are states such that $dom(s) \subseteq dom(s')$, e is any expression and c is canonical. The valid instances of the relation are inductively defined by a set of rules listed in the appendix. They use the sequentiality convention employed in the definition of SML [MTH90], which states that a rule of the form

$$\frac{e_1 \Downarrow c_1 \qquad e_2 \Downarrow c_2 \qquad \cdots \qquad e_n \Downarrow c_n}{e \Downarrow c}$$

is an abbreviation for

$$\frac{\langle s_1, e_1 \rangle \Downarrow \langle s_2, c_1 \rangle \qquad \langle s_2, e_2 \rangle \Downarrow \langle s_3, c_2 \rangle \qquad \cdots \qquad \langle s_n, e_n \rangle \Downarrow \langle s_{n+1}, c_n \rangle}{\langle s_1, e \rangle \Downarrow \langle s_{n+1}, c \rangle}$$

Note that, as in SML, the evaluation strategy is call-by-value.

2.2 The fragment of SML

The fragment of Core SML [MTH90] that we consider here is monomorphic, has no exception mechanism, and only has globally declared datatypes. The pattern matching and the local value declarations, which are the distinctive features of this fragment compared to the λ-calculus λML, are handled exactly as in the full Core SML. The commands for manipulating the state are the same in the fragment and in λML. The fragment we consider has products instead of pattern rows. This is a technical simplification, and it is easy to see how the argument advanced in this paper can be extended to cover expression rows.

The pattern matching and local declarations require separate syntactic categories. Values are the results of evaluating expressions, and only they are bound in local declarations and matched against a pattern. This reflects the call-by-value nature of evaluation in Standard ML. We have the following syntactic categories, called phrases, in the fragment:

- Expressions *exp*
- Matches *match*, which are lists of match rules.
- Match rules *mrule*, which associate an expression to a pattern. The expression is evaluated if the given value matches the pattern.
- Declarations *dec*, which associate values to variables.
- Value bindings *valbind*, which perform the binding in declarations.
- Patterns *pat*, which are matched against a value.

The grammar for raw SML phrases is as follows:

$$
\begin{array}{rcl}
exp & ::= & (exp, exp) \mid \texttt{true} \mid \texttt{false} \mid () \mid var \mid \texttt{ref}\, exp \mid \texttt{in}^{\kappa}(exp) \mid !exp \\
 & & \mid \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \mid exp\ exp \mid exp\ \texttt{id}\ exp \mid exp\colon ty \mid \texttt{fn}\colon ty\ \ match \\
match & ::= & mrule\ \langle\ \mid match\rangle \\
mrule & ::= & pat \Rightarrow exp \\
dec & ::= & val\ valbind \\
 & & \mid \texttt{local}\ dec\ \texttt{in}\ dec\ \texttt{end} \\
 & & \mid dec\ \langle ; \rangle\ dec \\
valbind & ::= & pat = exp\ \langle \texttt{and}\ valbind\rangle \\
 & & \mid \texttt{rec}\ valbind \\
pat & ::= & \_ \mid \texttt{true} \mid \texttt{false} \mid () \mid \texttt{ref}\ pat \mid \texttt{in}^{\kappa}(pat) \\
 & & var \mid (pat, \ldots, pat) \mid pat\colon ty \mid var\ \langle ty\rangle\ \texttt{as}\ pat
\end{array}
$$

Now we turn to the static semantics, i.e. the typing assignments. The types are the same as in the λ-calculus. The absence of polymorphism in the SML fragment makes it possible to simplify the judgements considerably. As in the λ-calculus, let a context Γ be a finite function from variables to types. We have the following kinds of judgements:

$\Gamma \vdash exp\colon \sigma$	The expression *exp* has type σ in context Γ.
$\Gamma \vdash mrule\colon \sigma$	The match rule *mrule* yields a value of type σ if matched in context Γ.
$\Gamma \vdash dec\colon \Delta$	The variables in the declaration *dec* form a context Δ with respect to the context Γ.
$\Gamma \vdash valbind\colon \Delta$	The value bindings in *valbind* yield the context Δ containing the new variables in *valbind*.
$\Gamma \vdash pat\colon (\Delta, \sigma)$	The successful matching of pattern *pat* against a value produces an expression of type σ and adds the variables contained in Δ to the evaluation context Γ.

Because the judgements are a restriction of those given in the definition of Standard ML [MTH90], we omit the details. An expression is called *closed* if $\emptyset \vdash exp\colon \sigma$, where $\emptyset$ denotes the empty context.

The canonical expressions of the λ-calculus form a subset of all expressions. This is necessary because the evaluation of an application works by first reducing the argument to a canonical expression and then substituting this expressions textually into the body of the function. The pattern matching in SML renders a similar definition of the evaluation relation impossible. A function application is evaluated not by substitution, but by matching the argument against the pattern the function specifies and evaluating the selected expression. Hence the result of evaluating a function is a closure, i.e. it contains both the body of the function and the values associated to all free variables in the function. As a consequence of these differences we call the result of an evaluation of a phrase a *value* and not a canonical expression.

The values are given by the grammar:

$$v ::= a \mid \texttt{true} \mid \texttt{false} \mid () \mid (v_1, v_2) \mid \texttt{in}^{\kappa}(v) \mid (match, E, E')$$

where E and E' are environments, i.e. finite functions from variables to values. We will write $(x_i := v_i)$ for such an environment. In the closure $(match, E, E')$ the environment E' contains the values for the variables that are used in recursive function calls and E contains the value for all other variables. We have the following kinds of judgements:

- $s, E \vdash exp \Downarrow v, s'$: The expression M evaluates to the value v in the environment E.
- $s, E, v \vdash match \Downarrow v', s'$: The match *match* is tested against the value v, yielding v' as a result.
- $s, E, v \vdash pat \Downarrow E, s'$: The pattern *pat* is matched against value v and added into the environment.
- $s, E \vdash dec \Downarrow E', s'$: The declaration *dec* is evaluated, resulting in a new declaration E'.
- $s, E \vdash valbind \Downarrow E', s'$: The binding in *valbind* is added to E.

Here s and s' are the states before and after evaluation, where a *state* is a finite function from addresses to values. The rules for generating these judgements are listed in the appendix. They are just as in [MTH90] (and employ the sequentiality convention mentioned above). In full SML, evaluation of pattern matches can raise exceptions. Here we just assume a restriction on expressions so that every match succeeds exactly for one pattern.

2.3 The translations between the languages

The translation from the λ-calculus into SML has to express all selectors like `fst` and `snd` by pattern matching. The translation in the other direction is essentially a compilation of pattern matching into the λ-calculus. Both translations are compositional: the translation of a phrase is defined in terms of the translations of its subphrases. The translation of λ-calculus expressions into SML expressions, denoted by $(-)^M$, is given in Table 1 and is quite straightforward. The translation in the other direction, denoted by $[\![-]\!]$, is given in Table 2 and is more involved. The translations $[\![exp]\!]$ and $[\![match]\!]$ of expressions and matches are λML expressions. The translations $[\![valbind]\!]$ and $[\![dec]\!]$ of value bindings and declarations are *substitutions*, σ, which are finite functions from identifiers to canonical λML expressions, indicated by $[c_i/x_i]$. The translation of a pattern is a tuple $(cond, \sigma)$. The first component *cond* is a boolean expression in λML that characterizes the condition a value has to satisfy to be successfully matched against the pattern. The second component is a substitution that describes how the variables declared by the pattern *pat* are obtained from the value matched against the pattern. The variable *arg* occurring in σ represents this value. The clause for matches $pat_1 \;\texttt{=>}\; exp_1|\cdots|pat_n \;\texttt{=>}\; exp_n$ describes a λ-term that tests the conditions $cond_1$ to $cond_n$ to select the succeeding pattern pat_i and returns the expressions exp_i with the substitution σ_i applied. A value binding *val pat* $=$ *exp* is translated into a substitution that binds the variables declared in the substitution originating from the translation of the pattern *pat* to the values obtained by performing the match.

As an example, consider the SML-function

$$\texttt{fn} : \texttt{bool} \times \texttt{bool}.(\texttt{true}, x) \;\texttt{=>}\; x \mid (\texttt{false}, y) \;\texttt{=>}\; \texttt{not}(y)\ .$$

The translation of the first pattern is the tuple $(\texttt{fst}(arg), [\texttt{snd}(arg)/x]$, and the translation of the second pattern is $(\texttt{not}(\texttt{fst}(arg)), [\texttt{snd}(arg)/y])$. Hence the translation of this function into λML is the function

$$\lambda arg\colon \texttt{bool} \times \texttt{bool}.\texttt{if}\ \texttt{fst}(arg)\ \texttt{then}\ \texttt{snd}(arg)\ \texttt{else}\ \texttt{not}(\texttt{snd}(arg))\ .$$

We extend the translation $[\![-]\!]$ simultaneously to values and environments as follows: for a value v, $[\![v]\!]$ is a canonical λML expression defined by induction on the structure of v (details omitted); for an environment $E = (x_i := v_i)$, $[\![E]\!]$ is defined to be the substitution $[[\![v_i]\!]/x_i]$. Finally SML states $s = \{a_i \mapsto v_i\}$ are translated into λML states by defining $[\![s]\!] = \{a_i \mapsto [\![v_i]\!]\}$, and similarly for the action of $(-)^M$ on λML states.

Both translations preserve strong typing, i.e. when an expression e is well-formed in context Γ, then its translation into the other language is well-formed as well. Moreover, both translation are *sound*, i.e. they preserve the validity of evaluation judgments. This statement becomes meaningful only after the addition of judgements about the evaluation of substitutions into the λML-calculus. The SML-judgements for the evaluation of patterns, declarations and value bindings correspond to judgement of this form in the λML-calculus. These properties can be shown by an induction over the derivation of the judgements.

The translation from SML into λML justifies why we omitted sum types and alternates in the definition of recursive datatypes. The construction of a λ-term that describes the selection of a succeeding match becomes significantly more complicated if these constructions are added. The proof that the translation between the calculi with these additional features are fully abstract follows the same line as the one for the calculus used in this paper. To simplify the exposition, we decided therefore not to consider such an extended calculus here.

3 Applicative bisimulation

We aim to show that the two translations both preserve and reflect the evaluation behaviour of expressions in all program contexts. We will do this by showing that $(-)^M$

Types

$$
\begin{array}{rcl}
(\kappa_i)^M & = & \text{type } \kappa_i \text{ in} \\
& & \quad \texttt{datatype}\ \ \kappa_1 = \ \ \texttt{in}^{\kappa_1}\ \texttt{of}\ \sigma_1 \\
& & \qquad\qquad\qquad\qquad \vdots \\
& & \qquad\quad \texttt{and}\ \ \kappa_n = \ \ \texttt{in}^{\kappa_n}\ \texttt{of}\ \sigma_n \\
(\texttt{unit})^M & = & \texttt{unit} \\
(\texttt{bool})^M & = & \texttt{bool} \\
(\sigma_1 \times \sigma_2)^M & = & (\sigma_1)^M \times (\sigma_2)^M \\
(\sigma_1 \to \sigma_2)^M & = & (\sigma_1)^M \to (\sigma_2)^M \\
(\sigma\ \texttt{ref})^M & = & (\sigma)^M\ \texttt{ref}
\end{array}
$$

Expressions

$$
\begin{array}{rcl}
(x)^M & = & x \\
(\texttt{in}^{\kappa_i}(e))^M & = & \texttt{in}^{\kappa}\texttt{i}((\texttt{e})^{\texttt{M}}) \\
(\texttt{out}(e))^M & = & \texttt{let val in}^{\kappa}(x) = (e)^M\ \texttt{in}\ x\ \texttt{end} \\
(())^M & = & () \\
(e_1 := e_2)^M & = & (e_1)^M := (e_2)^M \\
(\texttt{true})^M & = & \texttt{true} \\
(\texttt{false})^M & = & \texttt{false} \\
(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)^M & = & (\texttt{fn true: bool} \Rightarrow (e_2)^M \mid \texttt{false: bool} \Rightarrow (e_3)^M)(e_1)^M \\
(e_1 = e_2)^M & = & (e_1)^M = (e_2)^M \\
((e_1, e_2))^M & = & ((e_1)^M, (e_2)^M) \\
(\mathsf{Fst}(e))^M & = & \texttt{let val}(x, \_) = (e)^M\ \texttt{in}\ x\ \texttt{end} \\
(\mathsf{Snd}(e))^M & = & \texttt{let val}(\_, x) = (e)^M\ \texttt{in}\ x\ \texttt{end} \\
(\lambda x^{\sigma}.e)^M & = & \texttt{fn}\ x\texttt{:}\,(\sigma)^M \Rightarrow (e)^M \\
(e_1 e_2)^M & = & (e_1)^M (e_2)^M \\
(\texttt{rec}\ f(x) =^{\sigma} e_1\ \texttt{in}\ f(e_2))^M & = & \texttt{let val rec}\ f = \texttt{fn}\ x\texttt{:}\,\sigma \Rightarrow (e_1)^M\ \texttt{in}\ f((e_2)^M)\ \texttt{end} \\
(\texttt{ref}(e))^M & = & \texttt{ref}((e)^M) \\
(!e)^M & = & !(e)^M
\end{array}
$$

Table 1: Translation from λ-calculus to SML

The λ-expression $\pi_i(e)$ is an abbreviation for $\mathtt{fst}^i(\mathtt{snd}(e))$.

$$\begin{array}{rcl}
[\![(exp_1, exp_n)]\!] & = & ([\![exp_1]\!], \ldots, [\![exp_n]\!]) \\
[\![\mathtt{true}]\!] & = & \mathtt{true} \\
[\![\mathtt{false}]\!] & = & \mathtt{false} \\
[\![()]\!] & = & () \\
[\![\mathtt{in}^\kappa(exp)]\!] & = & \mathtt{in}^\kappa([\![exp]\!]) \\
[\![var]\!] & = & var \\
[\![!exp]\!] & = & ![\![exp]\!]
\end{array}$$

$$\frac{[\![dec]\!] = (t_1/x_1, \ldots, t_n/x_n)}{[\![\mathtt{let}\ dec\ \mathtt{in}\ exp\ \mathtt{end}]\!] = (\lambda x_n^{\sigma_1}.\cdots.(\lambda x_n.[\![exp]\!])[\![t_n]\!]\cdots)[\![t_1]\!]}$$

$$\begin{array}{rcl}
[\![exp\ exp]\!] & = & [\![exp]\!][\![exp]\!] \\
[\![\mathtt{fn}\colon ty\ match]\!] & = & \lambda arg\colon [\![ty]\!].[\![match]\!]
\end{array}$$

$$\frac{[\![mrule]\!] = (cond, exp)}{[\![mrule\ \langle\ \mid\ match\rangle]\!] = \langle \mathtt{if}\ cond\ \mathtt{then}\rangle\ exp\ \langle \mathtt{else}\ [\![match]\!]\rangle}$$

$$\frac{[\![pat]\!] = (cond, \sigma)}{[\![pat \Rightarrow exp]\!] = (cond, [\![exp]\!]\sigma)}$$

$$\begin{array}{rcl}
[\![val\ valbind]\!] & = & [\![valbind]\!] \\
[\![\mathtt{local}\ dec_1\ \mathtt{in}\ dec_2\ \mathtt{end}]\!] & = & [\![dec_2]\!][[\![dec_1]\!]] \\
[\![dec_1; dec_2]\!] & = & [\![dec_1]\!] + [\![dec_2]\!][[\![dec_1]\!]]
\end{array}$$

$$\frac{[\![pat]\!] = (cond, \sigma)}{[\![pat = exp]\!] = \sigma[[\![exp]\!]/arg]}$$

$$\frac{[\![pat = exp]\!] = \sigma \qquad [\![valbind]\!] = \sigma'}{[\![pat = exp\ \mathtt{and}\ valbind]\!] = \sigma \oplus \sigma'}$$

$$\frac{[\![valbind]\!] = [\lambda x_i\colon \sigma_i.t_i/f_i]}{[\![\mathtt{rec}\ valbind]\!] = [\lambda x_i\colon \sigma_i.\pi_i \mathtt{rec}\ f(arg) = (t_i[\pi_i(f)/f_i][\pi_i(x)/x_i])\ \mathtt{in}\ f(x_i)/f_i]}$$

$$[\![\_]\!] \quad = \quad (\mathtt{true}, \{\})$$

$$\begin{array}{rcl}
[\![\mathtt{true}]\!] & = & (arg, \{\}) \\
[\![\mathtt{false}]\!] & = & (\mathtt{not}(arg), \{\}) \\
[\![()]\!] & = & (\mathtt{true}, ()/arg) \\
[\![var]\!] & = & (\mathtt{true}, [var/arg])
\end{array}$$

Table 2: Translation from SML to λ-calculus

$$\frac{[\![pat_i]\!] = (cond_i, E_i)}{[\![pat_1, \ldots, pat_n]\!] = (\bigwedge cond_i[\pi_i(arg)/arg], \oplus E_i[\pi_i(arg)/arg])}$$

$$\frac{[\![pat]\!] = (cond, E)}{[\![\mathtt{in}^{\kappa}(pat)]\!] = (cond, E)[\mathtt{out}(arg)/arg]}$$

$$\frac{[\![pat]\!] = (cond, E)}{[\![\mathtt{ref}(pat)]\!] = (cond, E)[!(arg)/arg]}$$

$$[\![pat\colon ty]\!] = [\![pat]\!]$$

$$\frac{[\![pat]\!] = (cond, E)}{[\![var\langle : ty\rangle \ \mathtt{as}\ pat]\!] = (cond, E \oplus var/arg)}$$

Table 2 (continued): Translation from SML to λ-calculus

and $[\![-]\!]$ are mutually inverse[1] up to *observational equivalence*, written $\approx_{obs}$. By definition, $e_1 \approx_{obs} e_2$ holds in λML if for all closing boolean contexts $C[-]$ and states s, $\exists s_1.\langle s, C[e_1]\rangle \Downarrow \langle s_1, \mathtt{true}\rangle$ iff $\exists s_2.\langle s, C[e_2]\rangle \Downarrow \langle s_2, \mathtt{true}\rangle$. The definition of $\approx_{obs}$ for SML is similar. The quantification over all contexts makes it very difficult indeed to establish the required properties of the translations directly from the definition of $\approx_{obs}$. Instead, we proceed indirectly by giving co-inductively defined notions of program equivalence for λML and SML, called *applicative bisimilarity* and written $\approx_{app}$. Applicative bisimilarity implies observational equivalence and it is much easier to establish the (equational) properties of $\approx_{app}$ needed to show that the translations between λML and Standard ML are mutually inverse.

The difficult part in extending existing definitions of applicative bisimilarity for pure functional languages [AO93, How89, Gor93] to the languages studied here is the handling of the states involved in the evaluation relation. We were guided to the definition given below by analogy with the co-inductive characterization of equality in recursively defined domains given in [Pit94], applied to the domains needed for a denotational semantics of the λ-calculus λML. We explain the definition first for λML and adapt it afterwards to the fragment of SML.

3.1 Simulation for the λ-calculus λML

The relation of applicative similarity is defined as the greatest fixed point of a certain monotone operator $R \mapsto \overline{R}$ on relations between closed expressions. We start by giving the definition of $\overline{R}$ on canonical expressions. Afterwards we extend the definition of $\overline{R}$ to all closed expressions. Because it is not a priori clear that the extension of $\overline{R}$ to arbitrary expressions coincides with $\overline{R}$ on canonical expressions, we use a different symbol $\langle R\rangle$ for the relation $\overline{R}$ on canonical expressions at the moment.

[1] Modulo a restriction on patterns in the SML expressions arising from the fact that for simplicity we are avoiding exception handling: see Section 4.

Definition 1 *Given any family of relations* $R = (R_\sigma \subseteq Exp_\sigma \times Exp_\sigma \mid \sigma \in Types)$ *between closed* λML *expressions, let the family of relations* $\langle R \rangle_\sigma$ *between canonical expressions of type* σ *be inductively generated by the following rules:*

- $c \langle R \rangle_\sigma c'$ *if* $c \equiv c'$ *and* σ *is either* `unit`, `bool` *or a reference type.*
- $(c_1, c_2)\langle R \rangle_{\sigma_1 \times \sigma_2}(c_1', c_2')$ *if* $c_1 \langle R \rangle_{\sigma_1} c_1'$ *and* $c_2 \langle R \rangle_{\sigma_2} c_2'$.
- $\lambda x{:}\,\sigma.M \ \langle R \rangle_{\sigma \Rightarrow \sigma'} \ \lambda x{:}\,\sigma.M'$ *if for all canonical expressions* c *of type* σ*, we have* $M[c/x] \ R_{\sigma'} \ M'[c/x]$.
- $\mathsf{in}^\kappa(c) \ \langle R \rangle_\kappa \ \mathsf{in}^\kappa(c')$ *if* $c \ \langle R \rangle_\sigma \ c'$ *and* $decl(\kappa) = \sigma$.

For any two states s_0 and s_1, we will write $s_0\langle R \rangle s_1$ to mean that s_0 and s_1 have the same domain and for all addresses a in that domain $s_0(a) \ \langle R \rangle_\sigma \ s_1(a)$ (where σ `ref` is the type of a).

We extend the definition of $\langle R \rangle_\sigma$ to a relation between all closed expressions of type σ by requiring that whenever the first expression reduces to a canonical expression, the second does so, and that the resulting states respect the relation $\langle R \rangle_\sigma$. This extension is defined as follows:

Definition 2 *With* R *as in Definition 1, define another family* $\overline{R}$ *of relations between closed* λML *expressions of equal type as follows:*

$$
\begin{array}{ll}
M \overline{R_\sigma} M' \ \textit{iff} & \forall s_0, s_1, c \\
 & \langle s_0, M \rangle \Downarrow \langle s_1, c \rangle \Rightarrow \exists s_1', c'. \\
 & \langle s_0, M' \rangle \Downarrow \langle s_1, c' \rangle \ \textit{and} \ c \ \langle R \rangle_\sigma \ c' \ \textit{and} \ s_1 \ \langle R \rangle \ s_1' \ .
\end{array}
$$

The operator $R \mapsto \overline{R}$ is monotone and hence we can define a relation $\sqsubseteq$, called *applicative similarity*, as the greatest fixed point of this operator. As usual, $\sqsubseteq$ is in fact greatest amongst the post-fixed points, i.e. those R satisfying $R \subset \overline{R}$; such R are called applicative simulations. We extend the relation $\sqsubseteq$ to a relation between open expressions of the same type by defining $M \sqsubseteq M'$ to hold iff $M[c_i/x_i] \sqsubseteq M'[c_i/x_i]$ holds whenever c_i are canonical expressions and $M[c_i/x_i]$ and $M'[c_i/x_i]$ are closed. One can show easily that this extended relation $\sqsubseteq$ is a partial order and that for all canonical expressions c and c', $c\langle \sqsubseteq \rangle c'$ holds iff $c \sqsubseteq c'$ is satisfied. Applicative *bisimilarity*, written $\approx_{app}$, is defined as the symmetrization of $\sqsubseteq$: $M \approx_{app} M'$ iff $M \sqsubseteq M'$ and $M' \sqsubseteq M$.

The most important property of the relation $\sqsubseteq$ is that it is a pre-congruence. By this we mean that for all contexts $C[-]$, we have $C[M] \sqsubseteq C[N]$ whenever $M \sqsubseteq N$. The proof of this property is rather difficult. Howe [How89] describes a method for showing pre-congruence that has become well established: for example, it is used for a λ-calculus with non-determinism in [Ong93] and a λ-calculus with recursive datatypes and input/output in [Gor93]. One proceeds by defining a relation $\sqsubseteq^*$ that is easily seen to be a pre-congruence and includes $\sqsubseteq$; the inverse inclusion is then shown by a co-inductive argument. Here we have to adapt this method to cope with the presence of states in the operational semantics. We will omit the detailed proofs and sketch only the structure of the argument.

Definition 3 *The relation* $\sqsubseteq^*$ *between two well-formed expressions of the same type is defined by induction over the structure of expressions as follows:*

- $x \sqsubseteq^* M$ *whenever* $x \sqsubseteq M$.

- *For every operator τ of arity n, $\tau(M_1,\ldots,M_n) \sqsubseteq^* N$ whenever there exist $M_1',\ldots,M_n'$ such that $M_i \sqsubseteq^* M_i'$ for all $1 \leq i \leq n$ and $\tau(M_1',\ldots,M_n') \sqsubseteq N$. (Similarly for variable binding operators.)*

First, one establishes some properties of the relation $\sqsubseteq^*$.

Lemma 4 (i) *If $M \sqsubseteq^* M'$ and $M' \sqsubseteq M''$, then $M \sqsubseteq^* M''$.*

(ii) *$M \sqsubseteq^* M$.*

(iii) *$M \sqsubseteq M'$ implies $M \sqsubseteq^* M'$.*

(iv) *$\sqsubseteq^*$ is a congruence relation.*

(v) *If $M \sqsubseteq^* M'$ and $c \sqsubseteq^* c'$, then $M[c/x] \sqsubseteq^* M'[c'/x]$.*

The crucial part of the congruence proof is the following proposition,which is proved by induction over the derivation of $\langle s_0, M\rangle \Downarrow \langle s_1, c\rangle$.

Proposition 5 *Let M be any closed expressions of type σ. Then whenever $\langle s_0, M\rangle \Downarrow \langle s_1, c\rangle$ then for all closed expressions M' of type σ and states s_0' such that $M \sqsubseteq^* M'$, $s_0 \sqsubseteq^* s_0'$ and the addresses of M' are in the domain of s_0', there exists a canonical expression c' and a state s_1' such that $\langle s_0', M'\rangle \Downarrow \langle s_1', c'\rangle$ and $c \sqsubseteq^* c'$ and $s_1 \sqsubseteq^* s_1'$.*

This is sufficient to show

Proposition 6 *$M \sqsubseteq^* M'$ iff $M \sqsubseteq M'$.*

Proof It follows from Proposition 5 that $\sqsubseteq^*$ restricted to closed expressions is an applicative simulation, i.e. is a post-fixed point of the operation $R \mapsto \overline{R}$. Since the relation $\sqsubseteq$ is the greatest such, we have that $M \sqsubseteq^* M'$ implies $M \sqsubseteq M'$. The reverse implication is just Lemma 4(iii). □

The pre-congruence property of applicative similarity immediately yields

Corollary 7 *$M \approx_{app} M'$ implies $M \approx_{obs} M'$.*

However, $\approx_{app}$ does not coincide with $\approx_{obs}$, unlike the case for deterministic, pure functional languages. Partly this is due to the fact that in Definition 2 related states are required to have equal domains. As a consequence if $M \approx_{app} M'$ holds then evaluation of M and M' creates the same number of fresh addresses for local references, whereas such a property of address creation need not hold when $M \approx_{obs} M'$. (For example, $(\lambda x^{\texttt{bool ref}}.())\texttt{ref(true)}$ is observationally equivalent to $()$, but is not applicatively bisimilar to it.) This particular defect is remedied by the more refined notions of applicative equivalence considered in [PS93, PPS94]. We do not need such refinements to establish the full abstraction result of this paper. However, at the moment there is no known notion of applicative equivalence that coincides with observational equivalence for languages with dynamically created local state.

3.2 Simulation for SML

Now we define the applicative bisimulation for the fragment of SML. The idea behind the definition is the same, but the technical details are more complex due to the various syntactic categories of the language. Because the expression $M[c/x]$ in the λ-calculus λML corresponds to the pair $(E + (x := v), exp)$ in SML, we have to consider (environment, expression)-pairs. For this purpose we define a *generalized expression* to be either (E, exp), $(E + v, match)$ or (E, dec), where exp is an expression, v a value, $match$ a match, and dec a declaration. Such a generalized expression is called *closed* if $\emptyset \vdash E\colon\Gamma$ and exp, $match$ or dec are well-formed in context Γ. We define the applicative simulation for these generalized expressions.

Definition 8 *Given any type-indexed family R of relations between closed generalized expressions of equal type, a family of relations $\langle R\rangle_\sigma$ between values of each type σ is inductively defined by the following rules:*

- $v_1 \; \langle R\rangle_\sigma \; v_2$ *if* $v_1 \equiv v_2$ *and* σ *is* `bool`*,* `unit`*, or a reference type.*
- $(v_1, v_2)\langle R\rangle_{\sigma_1\times\sigma_2}(v_1', v_2')$ *if* $v_1\langle R\rangle_{\sigma_1}v_1'$ *and* $v_2\langle R\rangle_{\sigma_2}v_2'$*.*
- $\mathtt{in}^\kappa(v_1)\langle R\rangle_\kappa\mathtt{in}^\kappa(v_2)$ *if* $v_1 \; \langle R\rangle_\sigma \; v_2$*, where* $decl(\kappa) = \sigma$*.*
- $(match_1, E_1, E_1')\langle R\rangle_{\sigma\Rightarrow\sigma'}(match_2, E_2, E_2')$ *if for all values v of type σ, we have* $(E_1, E_1' + v, match_1) \; R \; (E_2, E_2' + v, match_2)$*.*

The relation $\langle R\rangle$ is extended to states as for λML: for any two states s_0 and s_1, we write $s_0\langle R\rangle s_1$ if s_0 and s_1 have equal domains and are related argument-wise by $\langle R\rangle$. We also extend $\langle R\rangle$ to a relation between environments, $E_1\langle R\rangle E_2$, in exactly the same way. Using these definitions, the extension of $\langle R\rangle$ to all closed generalized expressions is defined as follows:

Definition 9 *Let R be as in Definition 8. We define another such family of relations $\overline{R}$ as follows:*

(i) $(E_1, exp_1) \; \overline{R} \; (E_2, exp_2)$ *iff* $\forall s_0, s_1, v_1$
$s_0, E_1 \vdash exp_1 \Downarrow v_1, s_1$ *implies* $\exists v_2, s_2$ *such that*
$s_0, E_2 \vdash exp_2 \Downarrow v_2, s_2$ *and* $s_1\langle R\rangle s_2$ *and* $v_1 \; \langle R\rangle_\sigma \; v_2$ *(where σ is the type of v_1, v_2).*

(ii) $(E_1 + v_1, match_1) \; \overline{R} \; (E_2 + v_2, match_2)$ *iff* $\forall s_0, s_1, v_1$
$s_0, E_1, v_1 \vdash match_1 \Downarrow v_1, s_1$ *implies* $\exists v_2, s_2$ *such that*
$s_0, E_2, v_2 \vdash match_2 \Downarrow v_2, s_2$ *and* $s_1\langle R\rangle s_2$ *and* $v_1 \; \langle R\rangle_\sigma \; v_2$ *(where σ is the type of v_1, v_2).*

(iii) $(E_1, dec_1) \; \overline{R} \; (E_2, dec_2)$ *iff* $\forall s_0, s_1, E_1'$
$s_0, E_1 \vdash dec_1 \Downarrow E_1', s_1$ *implies* $\exists E_2', s_2$ *such that*
$s_0, E_2 \vdash dec_2 \Downarrow E_2', s_2$ *and* $E_1'\langle R\rangle E_2'$ *and* $s_1\langle R\rangle s_2$*.*

Let $\sqsubseteq$ be the greatest fixed point of the monotone operator $R \mapsto \overline{R}$. Then $\sqsubseteq$ is extended to open generalized expressions by defining $(E_1, exp_1) \sqsubseteq (E_2, exp_2)$ to hold iff for all values v_i such that $(E_1 + (x_i := v_i), exp_1)$ and $(E_2 + (x_i := v_i), exp_2)$ are closed, we have $(E_1 + (x_i := v_i), exp_1) \sqsubseteq (E_2 + (x_i := v_i), exp_2)$. The relation of applicative bisimilarity for SML is the symmetrization of this relation: $(E_1, exp_1) \approx_{app} (E_2, exp_2)$ iff $(E_1, exp_1) \sqsubseteq (E_2, exp_2)$ and $(E_2, exp_2) \sqsubseteq (E_1, exp_1)$.

A relation $\sqsubseteq^*$ is defined from $\sqsubseteq$ in much the same way as for the λ-calculus λML and the analogues of Lemma 4 and Proposition 5 can be established. Thus $\sqsubseteq$ coincides with $\sqsubseteq^*$ and the latter is a pre-congruence. As in the λ-calculus, the congruence property of the relation $\sqsubseteq$ yields immediately that SML applicative bisimilarity implies observational equivalence.

Theorem 10 *For all closed SML expressions exp_1, exp_2 (of equal type), $(\emptyset, exp_1) \approx_{app} (\emptyset, exp_2)$ implies $exp_1 \approx_{obs} exp_2$.*

4 The Equivalence

We are now in a position to establish the full abstraction properties of the translations between SML and the λ-calculus λML. We first show that if we translate an expression to the other calculus and back, we get a result that is applicatively bisimilar to the expression we started with. Because we are not considering exception handling in this paper, for this property to hold for the SML expressions we have to restrict to ones involving matches that succeed for exactly one pattern. For example, consider the function $exp = \texttt{fn true} \Rightarrow \texttt{true}$; we have $([\![exp]\!])^M = \texttt{fn}\ arg \Rightarrow \texttt{true}$, and hence $(exp)\texttt{false}$ raises an exception in SML, whereas $([\![exp]\!])^M\texttt{false}$ evaluates to $\texttt{true}$.

Theorem 11 *For any λML expression e, we have $[\![(e)^M]\!] \approx_{app} e$. For any generalized SML expression (E, exp) with exp satisfying the restriction on matches mentioned above, we have $([\![E, exp]\!])^M \approx_{app} (E, M)$, where $[\![E, exp]\!]$ denotes the λML expression obtained by applying the substitution $[\![E]\!]$ to the expression $[\![exp]\!]$.*

Proof (sketch) The first statement is shown by induction over the structure of e. For the second statement, we show by induction over the structure of exp, that whenever all values v in E satisfy $([\![v]\!])^M \approx_{app} v$, then $([\![E, exp]\!])^M \approx_{app} (E, exp)$.

The induction steps make use of certain identities that hold up to applicative bisimilarity, which are easily established from the co-inductive definition of $\sqsubseteq$. For example, consider the λML expression $\texttt{fst}(e)$. The translation into SML and back yields the term $(\lambda arg^{\sigma_1 \times \sigma_2}.\texttt{fst}(arg))[\![(e)^M]\!]$. By induction hypothesis $[\![(e)^M]\!] \approx_{app} e$ holds. Hence by the congruence property of $\approx_{app}$ established in Section 3.1, we have $[\![(\texttt{fst}(e))^M]\!] \approx_{app} (\lambda arg^{\sigma_1 \times \sigma_2}.\texttt{fst}(arg))e$. Then $[\![(\texttt{fst}(e))^M]\!] \approx_{app} \texttt{fst}(e)$ follows from the identity

$$(\lambda arg^{\sigma_1 \times \sigma_2}.\texttt{fst}(arg))e \approx_{app} \texttt{fst}(e)$$

which is easily established from the fact that the preorder $\sqsubseteq$ inducing $\approx_{app}$ is a fixed point (indeed the greatest one) of the monotone operator $R \mapsto \overline{R}$ of Definition 2. □

The main result of the paper now follows from the fact that applicative bisimilarity implies observational equivalence, together with the above theorem and the general properties of the translations of compositionality and soundness with respect to evaluation.

Theorem 12 *The translations $[\![-]\!]$ and $(-)^M$ between the SML fragment and the λ-calculus λML are fully abstract, in the sense that they preserve and reflect observational equivalence:*

$$\begin{aligned} e_1 \approx_{obs} e_2 \quad &\text{iff} \quad (e_1)^M \approx_{obs} (e_2)^M \\ exp_1 \approx_{obs} exp_2 \quad &\text{iff} \quad [\![exp_1]\!] \approx_{obs} [\![exp_2]\!] \end{aligned}$$

(provided exp_1, exp_2 satisfy the restriction on matches mentioned above).

Proof Combining Corollary 7 with Theorems 10 and 11, we have that $[\![(e)^M]\!] \approx_{obs} e$ and $([\![exp]\!])^M \approx_{obs} exp$. From this it follows that it is sufficient to just prove that each translation preserves observational equivalence.

So suppose that $e_1 \approx_{obs} e_2$ in λML and that in SML for some boolean context $C[-]$ we have $s_0, \emptyset \vdash C[(e_1)^M] \Downarrow \mathtt{true}, s_1$. Then by the soundness property of $[\![-]\!]$, in λML we have $\langle [\![s_0]\!], [\![\emptyset, C[(e_1)^M]]\!] \rangle \Downarrow \langle [\![s_1]\!], [\![\mathtt{true}]\!] \rangle$. Now $[\![\mathtt{true}]\!] = \mathtt{true}$; and by compositionality of $[\![-]\!]$ and the fact that $\approx_{app}$ is a congruence, we have

$$[\![\emptyset, C[(e_1)^M]]\!] \equiv [\![C]\!][[\![(e_1)^M]\!]] \approx_{app} [\![C]\!][e_1] \approx_{app} [\![C]\!][e_2] \,.$$

Hence (by Proposition 5) for some s_2' $\langle [\![s_0]\!], [\![C]\!][e_2] \rangle \Downarrow \langle s_2', \mathtt{true} \rangle$. Then by the soundness property of $(-)^M$, in SML we have $([\![s_0]\!])^M, \emptyset \vdash ([\![C]\!][e_2])^M \Downarrow (\mathtt{true})^M, (s_2')^M$. Since $(\mathtt{true})^M = \mathtt{true}$, $([\![s_0]\!])^M \approx_{app} s_0$ and $([\![C]\!][e_2])^M \equiv ([\![C]\!])^M[(e_2)^M] \approx_{app} C[(e_2)^M]$, it follows that there is some s_2 with $s_0, \emptyset \vdash C[(e_2)^M] \Downarrow \mathtt{true}, s_2$. Thus by definition of observational equivalence we have that $(e_1)^M \approx_{obs} (e_2)^M$ in SML when $e_1 \approx_{obs} e_2$ in λML. The argument that $[\![-]\!]$ preserves observational equivalence is similar. $\square$

5 Conclusions

This paper describes a translation from a fragment of Standard ML to a λ-calculus with reference types, the main feature of which is a compilation of SML's use of pattern matching into λ-terms. We proved that the translation is fully abstract, i.e. that it preserves and reflects observational equivalence. The proof of this property is surprisingly difficult, because it is hard to reason directly about observational equivalence for programming languages such as SML that involve dynamic creation of local storage for higher order functions. A co-inductively defined notion of applicative bisimilarity is used here to obtain the result.

The notion of applicative similarity developed as a means to an end in this paper seems interesting in its own right. It implies observational equivalence, but not vice versa. A closely related notion, which validates more observational equivalences, is developed in [PPS94]. It remains an open problem to find a co-inductive characterization of observational equivalence for languages like SML that combine higher order functions and local state. Nevertheless, we believe that the existing notions of applicative bisimilarity, or refinements of them, may provide simpler methods for verifying program properties for languages like SML compared with denotational methods, or with reasoning directly about observational equivalence (as in [MT91, MT92], for example).

The fragment of Standard ML we have considered is monomorphic in order to avoid the known difficulties with mixing ML polymorphism with reference types—difficulties that are largely irrelevant to the concerns of this paper. For simplicity, we also excluded any exception-handling mechanism and alternate clauses in the definition of recursive datatypes from the fragment. We do not envisage any problem in extending the definition of applicative bisimilarity to cope with these features, although we have not considered this yet. Apart from anything else, such an extension would enable Standard ML's method for evaluating incomplete patterns to be treated.

References

[AO93] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[Gor93] A. Gordon. *Functional Programming and Input/Output.* PhD thesis, University of Cambridge, 1993. Also available as Technical Report No. 285.

[How89] D. J. Howe. Equality in lazy computation systems. In *Proc. 4th Annual Symp. Logic in Computer Science*, pages 198–203. IEEE Computer Society Press, 1989.

[MT91] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[MT92] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *Proc. 7th Annual Symp. Logic in Computer Science*, pages 186–197. IEEE Computer Society Press, 1992.

[Mil77] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Ong93] C.-H. L. Ong. Non-determinism in a functional setting–extended abstract. In *Proc. 8th Annual Symp. Logic in Computer Science, Montréal, Canada*, pages 275–286. IEEE Computer Society Press, 1993.

[PPS94] V. C. V. de Paiva, A. M. Pitts and I. D. B. Stark. A Monadic ML. In preparation.

[Pit94] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[PS93] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Proc. Int. Symp. on Math. Foundations of Computer Science*, pages 122–141. Lecture Notes in Computer Science No. 711, Berlin, 1993.

[Rie90] J. G. Riecke. A complete and decidable proof system for call-by-value equalities. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming, Warwick*, pages 20–31. Lecture Notes in Computer Science Vol. 443, Berlin, 1990.

A Appendix

First we list the judgements defining well-typed terms in λML as well as the rules for evaluation to canoncial form. Second, we present the evaluation rules for the fragment of SML.

A.1 Rules for λML

The rules for well-typed terms in λML are as follows:

$$\frac{}{\Gamma \vdash x\colon \Gamma(x)} \qquad \frac{\Gamma \vdash e\colon \sigma}{\Gamma \vdash \mathtt{in}^{\kappa}(e)\colon \kappa}\,(decl(\kappa = \sigma))$$

$$\frac{\Gamma \vdash e\colon \kappa}{\Gamma \vdash \mathtt{out}(e)\colon \sigma}\,(\sigma = decl(\kappa)) \qquad \frac{}{\Gamma \vdash ()\colon \mathtt{unit}}$$

$$\frac{\Gamma \vdash r\colon \sigma\;\mathtt{ref}}{\Gamma \vdash (r := e)\colon \mathtt{unit}} \qquad \frac{}{\Gamma \vdash \mathtt{true}\colon \mathtt{bool}}$$

$$\frac{}{\Gamma \vdash \mathtt{false}\colon \mathtt{bool}} \qquad \frac{\Gamma \vdash b\colon \mathtt{bool} \quad \Gamma \vdash e\colon \sigma \quad \Gamma \vdash e'\colon \sigma}{\Gamma \vdash (\mathtt{if}\; b\; \mathtt{then}\; e\; \mathtt{else}\; e')\colon \sigma}$$

$$\frac{\Gamma \vdash r\colon \sigma\;\mathtt{ref} \quad \Gamma \vdash r'\colon \sigma\;\mathtt{ref}}{\Gamma \vdash (r = r')\colon \mathtt{bool}} \qquad \frac{\Gamma \vdash e\colon \sigma' \quad \Gamma \vdash e'\colon \sigma'}{\Gamma \vdash (e, e')\colon \sigma \times \sigma'}$$

$$\frac{\Gamma \vdash e\colon \sigma \times \sigma'}{\Gamma \vdash \mathtt{fst}(e)\colon \sigma} \qquad \frac{\Gamma \vdash e\colon \sigma \times \sigma'}{\Gamma \vdash \mathtt{snd}(e)\colon \sigma'}$$

$$\frac{\Gamma, x\colon \sigma \vdash e\colon \sigma'}{\Gamma \vdash \lambda x^{\sigma}.e\colon \sigma \to \sigma'} \qquad \frac{\Gamma \vdash e\colon \sigma \to \sigma' \quad \Gamma \vdash e'\colon \sigma}{\Gamma \vdash ee'\colon \sigma'}$$

$$\frac{\Gamma, f\colon \sigma \to \sigma', x\colon \sigma \vdash e'\colon \sigma' \quad \Gamma \vdash e\colon \sigma}{\Gamma \vdash \mathtt{rec}\; f(x) =_{\sigma} e'\; \mathtt{in}\; f(e)\colon \sigma'} \qquad \frac{\Gamma \vdash e\colon \sigma}{\Gamma \vdash \mathtt{ref}(e)\colon \sigma\;\mathtt{ref}}$$

$$\frac{\Gamma \vdash r\colon \sigma\;\mathtt{ref}}{\Gamma \vdash !r\colon \sigma} \qquad \frac{}{\Gamma \vdash a^{\sigma}\colon \sigma\;\mathtt{ref}}$$

The rules for evaluation to canonical expression are as follows:

$$\frac{e \Downarrow c}{\mathtt{in}^{\kappa}(e) \Downarrow \mathtt{in}^{\kappa}(c)} \qquad \frac{e \Downarrow \mathtt{in}^{\kappa}(c)}{\mathtt{out}(e) \Downarrow c}$$

$$\frac{}{() \Downarrow ()} \qquad \frac{\langle s_1, r\rangle \Downarrow \langle s_2, a\rangle \quad \langle s_2, e\rangle \Downarrow \langle s_3, c\rangle}{\langle s_1, r := e\rangle \Downarrow \langle s_3 \cup \{a \mapsto c\}, ()\rangle}$$

$$\frac{}{\mathtt{true} \Downarrow \mathtt{true}} \qquad \frac{}{\mathtt{false} \Downarrow \mathtt{false}}$$

$$\frac{b \Downarrow \mathtt{true} \quad e \Downarrow c}{\mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' \Downarrow c} \qquad \frac{\langle s_1, e\rangle \Downarrow \langle s_2, a\rangle}{\langle s_1, !e\rangle \Downarrow \langle s_2, c\rangle}\,(c = s_2(a))$$

$$\frac{b \Downarrow \mathtt{false} \quad e' \Downarrow c}{\mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' \Downarrow c} \qquad \frac{r \Downarrow a \quad r' \Downarrow a}{r = r' \Downarrow \mathtt{true}}$$

$$\frac{r \Downarrow a \quad r' \Downarrow a'}{r = r' \Downarrow \mathtt{false}}\,(a \neq a') \qquad \frac{e \Downarrow c \quad e' \Downarrow c'}{(e, e') \Downarrow (c, c')}$$

$$\frac{e \Downarrow (c, c')}{\mathtt{fst}(e) \Downarrow c} \qquad \frac{e \Downarrow (c, c')}{\mathtt{snd}(e) \Downarrow c'}$$

$$\frac{}{\lambda x^{\sigma}.e \Downarrow \lambda x^{\sigma}.e} \qquad \frac{f \Downarrow \lambda x^{\sigma}.e' \quad e \Downarrow c \quad e'[c/x] \Downarrow c'}{fd \Downarrow c'}$$

$$\frac{}{a \Downarrow a} \qquad \frac{e \Downarrow c \quad e'[\lambda x^{\sigma}.\mathtt{rec}\ f(x) =_{\sigma} e'\ \mathtt{in}\ f(x)/f, c/x] \Downarrow c'}{\mathtt{rec}\ f(x) =_{\sigma} e'\ \mathtt{in}\ f(e) \Downarrow c'}$$

$$\frac{\langle s_1, e\rangle \Downarrow \langle s_2, c\rangle}{\langle s_1, \mathtt{ref}(e)\rangle \Downarrow \langle s_2 \oplus \{a \mapsto c\}, a\rangle}\,(a \notin dom(s_2))$$

A.2 Rules for SML

The rules for the dynamic semantics of the SML fragment are as follows. They follow closely the corresponding rules for core Standard ML in [MTH90]. We assume that every match will succeed exactly for one pattern. If this assumption is not made, an exception may occur in Standard ML, which has no counterpart in the λ-calculus. The sequentiality convention that was mentioned for the λ-calculus holds, too.

Expressions:

$$(value) \qquad \frac{}{E \vdash v \Downarrow v}$$

$$(var) \qquad \frac{}{E \vdash var \Downarrow v}(E(var) = v)$$

$$(prod) \qquad \frac{E \vdash exp_1 \Downarrow v_1 \qquad E \vdash exp_2 \vdash v_2}{E \vdash (exp_1, exp_2) \Downarrow (v_1, v_2)}$$

$$(let) \qquad \frac{E \vdash dec \Downarrow E' \qquad E + E' \vdash exp \Downarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Downarrow v}$$

$$(\texttt{in}) \qquad \frac{E \vdash exp \Downarrow v}{E \vdash \texttt{in}^{\kappa}(exp) \Downarrow in^{\kappa}(v)}$$

$$(\texttt{ref}) \qquad \frac{s, E \vdash exp \Downarrow v, s'}{s, E \vdash \texttt{ref}\ exp \Downarrow a, s' + \{a \mapsto v\}}(a \notin s')$$

$$(:=) \qquad \frac{s, E \vdash exp_1 \Downarrow a, s' \qquad s', E \vdash exp_2 \vdash v, s''}{s, E \vdash exp_1 := exp_2 \Downarrow (), s'' + \{a \mapsto v\}}$$

$$(match) \qquad \frac{E \vdash exp_1 \Downarrow (match, E', VE) \qquad E \vdash exp_2 \Downarrow v \qquad E' + VE, v \vdash match \Downarrow v'}{E \vdash exp_1 exp_2 \Downarrow v'}$$

$$(fn) \qquad \frac{}{E \vdash \texttt{fn}\ match \Downarrow (match, E, \{\})}$$

Matches:

$$(success) \qquad \frac{E, v \vdash mrule \Downarrow v'}{E, v \vdash mrule < |match > \Downarrow v'}$$

$$(next) \qquad \frac{E, v \vdash mrule \Downarrow \text{FAIL} \qquad E, v \vdash match \Downarrow v'/\text{FAIL}}{E \vdash mrule|match \Downarrow v'/\text{FAIL}}$$

Match Rules:

$$(success) \qquad \frac{E, v \vdash pat \Downarrow VE \qquad E + VE \vdash exp \Downarrow v'}{E, v \vdash pat \Rightarrow exp \Downarrow v'}$$

$$(FAIL) \qquad \frac{E, v \vdash pat \Downarrow \text{FAIL}}{E, v \vdash pat \Rightarrow exp \Downarrow \text{FAIL}}$$

Declarations:

$$(val) \qquad \frac{E \vdash valbind \Downarrow VE}{E \vdash \texttt{val}\ valbind \Downarrow VE}$$

$$(local) \qquad \frac{E \vdash dec_1 \Downarrow E_1 \qquad E + E_1 \vdash dec_2 \Downarrow E_2}{E \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Downarrow E_2}$$

$$(empty) \qquad \frac{}{E \vdash \quad \Downarrow \{\}}$$

$$(;) \qquad \frac{E \vdash dec_1 \Downarrow E_1 \qquad E + E_1 \vdash dec_2 \Downarrow E_2}{E \vdash dec_1 \langle ; \rangle dec_2 \Downarrow E_1 + E_2}$$

Value Bindings:

$$(=)\qquad \frac{E \vdash exp \Downarrow v \qquad E, v \vdash pat \Downarrow VE \qquad \langle E \vdash valbind \Downarrow VE' \rangle}{E \vdash pat = exp\ \langle \texttt{and}\ valbind \rangle \Downarrow VE \langle + VE' \rangle}$$

$$(\texttt{rec})\qquad \frac{E \vdash valbind \Downarrow VE}{E \vdash \texttt{rec}\ valbind \Downarrow VE}$$