Limits of ML-definability

Stefan Kahrs

University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings,
Edinburgh EH9 3JZ
United Kingdom
email: `smk@dcs.ed.ac.uk`

Abstract. It is well-known that the type system of ML is overly restrictive in its handling of recursion: certain intuitively sound *terms* do not pass ML's type-check. We formalise this intuition and show that the restriction is semantical: there are computable (semantical) *functions* which cannot be expressed by well-typed (syntactical) terms.

Keywords: definability, polymorphism, recursion, ML, completeness

1 Introduction and Outline

Are all computable functions definable in ML? One should think so, after all ML supports general recursion and it is easy to define natural numbers with all their primitive operations using ML's datatypes.

But what about computable functions on other types, for example function types and inductive datatypes? Is ML "complete" w.r.t. to those types as well? Can we invert Milner's slogan "well-typed programs don't go wrong" and type-check programs that always go right? Before we address this specific question we shall define a general notion of completeness in a more abstract context.

1.1 Type Systems for Labelled Transition Systems

We first look at a general notion of type system for labelled transition systems. This is explored at greater depth in [5]. The idea is to view the operational semantics of a programming language as a labelled transition system and define an appropriate notion of completeness on this abstract level. Moreover, imposing a type system on a programming language can be viewed as an operation on transition systems.

Definition 1.1. A *transition system* is a structure $(\mathrm{Sta}, \mathrm{Lab}, \mathrm{Tra})$ where

- Sta is a set of *states*,
- Lab is a set of *labels*, and
- $\mathrm{Tra} \subset \mathrm{Sta} \times \mathrm{Lab} \times \mathrm{Sta}$ is the *transition relation*.

A transition system is called *pointed* if there is a distinguished initial state ι.

As usual, we write $s \xrightarrow{l} s'$ for $(s, l, s') \in \mathrm{Tra}$; also, $s \xrightarrow{w} s'$ with $w \in \mathrm{Lab}^*$ is shorthand for: there exist states $s_0, \dots, s_n$ and labels $l_1, \dots, l_n$ with $s = s_0$, $s' = s_n$, $w = l_1 \cdots l_n$ and $s_{i-1} \xrightarrow{l_i} s_i$. If $s \xrightarrow{w} s'$ then s' is *reachable from* s; a state is *reachable* iff it is reachable from the initial state ι. We write $s \twoheadrightarrow s'$ for the reachability relation $\exists w \in \mathrm{Lab}^*.\, s \xrightarrow{w} s'$.

Definition 1.2. Given a transition system, a relation $\lesssim$ on states is a *simulation* if the following holds: If $s_1 \lesssim s_2$ then

$$\forall l \in \mathrm{Lab}, s_1' \in \mathrm{Sta}.\, (s_1 \xrightarrow{l} s_1' \;\supset\; \exists s_2' \in \mathrm{Sta}.\, (s_2 \xrightarrow{l} s_2' \wedge s_1' \lesssim s_2')).$$

A *bisimulation* is a simulation $\lesssim$ such that $\gtrsim$ is a simulation as well.

Here are a few useful observations on simulations and bisimulations: simulations are closed under (arbitrary) union and composition, hence the transitive closure of a simulation is a simulation and there is always a largest simulation. We write $\precapprox$ for the largest simulation in a TS (which is necessarily a preorder) and $\approx$ for the largest bisimulation (which is necessarily an equivalence). The reflexive transitive closure of a simulation is also a simulation, since the identity relation obviously is one. Bisimulations are also closed under inversion and hence under equivalence closure or partial equivalence closure. The symmetric interior of a simulation is a bisimulation whenever the TS is deterministic.

For pointed transition systems, one could say that they are complete iff every state is reachable. However, this tends to be a stronger condition than we are really interested in: we only want to reach states up to bisimilarity:

Definition 1.3. A state s is called *complete* iff $\forall s_1 \in \mathrm{Sta}.\, \exists s_1' \in \mathrm{Sta}.\, (s \twoheadrightarrow s_1' \wedge s_1 \precapprox s_1')$. It is called *bicomplete* iff $\forall s_1 \in \mathrm{Sta}.\, \exists s_1' \in \mathrm{Sta}.\, (s \twoheadrightarrow s_1' \wedge s_1 \approx s_1')$. A pointed TS is (bi-) complete iff its initial state ι is (bi-) complete.

For a corresponding notion of soundness we could declare a subset of states as error states with the idea of declaring a TS as sound if no error state is reachable and a state s as sound if no error state is reachable from s. In the presence of error states, the notion of completeness should be modified, i.e. we are only interested to reach each *sound* state (up to bisimilarity).

Definition 1.4. A *typed transition system* (short: TTS) is a structure $(A, B, :)$ such that A and B are pointed transition systems and $: \subseteq \mathrm{Lab}_A \times \mathrm{Lab}_B$. Given a TTS $(A, B, :)$ we define the associated *untyped transition system* $A : B$ as the pointed TS given as:

- the set of states $\mathrm{Sta}_A \times \mathrm{Sta}_B$,
- the set of labels $\mathrm{Lab}_A \times \mathrm{Lab}_B$,
- the initial state (ι_A, ι_B),
- the transition relation:
  $(X, Y) \xrightarrow{(x,y)} (X', Y') \iff x : y \wedge X \xrightarrow{x} X' \wedge Y \xrightarrow{y} Y'$.

It is beyond the scope of this paper to motivate the definition of TTS in depth [5]. However, the reader may observe in the course of this paper how the general definition neatly specialises to type systems for functional programming.

1.2 Completeness and ML

Traditionally [6], the operational semantics of ML is defined through a ternary relation $E \vdash e \to v$, meaning that the expression e evaluates in the environment E to the value v. We can turn this into a transition system M by taking environments as states and defining transitions as follows:

$$E \xrightarrow{(e,x)} E[x \mapsto v] \iff E \vdash e \to v$$

In other words, we can evaluate an expression and bind the resulting value to a variable. This corresponds very closely to the behaviour of an SML interpreter which (on top-level) evaluates *declarations* rather than expressions. We can turn M into a pointed TS by choosing the empty environment as initial state.

The structure of a type system for ML is similar: we have a similar ternary relation, with type environments instead of environments and types instead of values. The labels (e, x) are exactly the same. We can therefore turn a type system into a corresponding pointed transition system T and form a TTS $(M, T, :)$ where ":" is the equality relation. While M can be seen as the underlying untyped language, $M : T$ is the typed one in which we only evaluate well-typed expressions: each state in $M : T$ is a pair of a type environment and a (value) environment, and in each reachable state the types and values "fit" in a certain sense. One can identify various primitive error scenarios in ML which should not occur on run-time, e.g. using a number as a function. We can add corresponding transitions to error states whenever these scenarios occur — by the soundness of ML's type system these error states are unreachable. The completeness question is whether there are unreachable states in which the types and values still fit (i.e. the state is sound), but which cannot be simulated by reachable ones.

ML's type system limits the way a recursive function can be used in its own definition; although the function is polymorphic, the recursive calls cannot exploit this polymorphism.

```
fun f x = f f
```

This function declaration does not type-check in SML, although `f` is perfectly sound (for type $\alpha \to \beta$): it is the totally undefined function; moreover, the declaration would type-check in an extension of ML's type system that guarantees soundness [4]. But the example only shows that certain sound *function declarations* do not type-check. The corresponding value is still ML-definable:

```
fun g x = g x
```

which does type-check such that the value of `g` is equivalent to the value of `f` w.r.t. all function types, i.e. replacing one by the other in some environment E results in a bisimilar E' and thus this example does not show incompleteness.

Let us have a look at another example that makes use of these features and is limited by ML's approach to recursion in a more substantial way:

```
datatype 'a lift = One | An of 'a
(*  unlift: ('a lift) list -> 'a list *)
fun unlift [] = []
 |  unlift (An x::xs) = x::unlift xs
 |  unlift (One::xs) = unlift xs

datatype 'a lam =
    Var of 'a | App of 'a lam * 'a lam | Lam of ('a lift) lam
(*  fvars: 'a lam -> 'a list *)
fun fvars (Var x) = [x]
 |  fvars (App (t,u)) = fvars t @ fvars u
 |  fvars (Lam t) = unlift (fvars t)
```

The idea of the example is that the type `'a lam` is the type of λ-calculus terms with de Bruijn indices [2], where we leave the control over increasing and decreasing indices to the type system. For example, the named λ-term $\lambda x.\lambda y.x$ has a de Bruijn representation of $\lambda\lambda 1$ which is expressed as `Lam(Lam(Var (An One)))`. The function `fvars` is supposed to compute the list of free variables of a term. We can observe that this function definition does not typecheck in SML, because the last recursive call uses `fvars` in the type instance `('a lift) lam` $\to$ `('a lift) list`.

Thus we have a similar problem as in the earlier example, a sound function declaration that is rejected by ML's type system. The similarity ends here. This time there is no way to define an equivalent environment in SML and we shall later see why. The problem arises whenever we try to define a non-trivial recursive function on a recursive datatype with a non-regular recursive structure.

One could argue how useful these types are in practice, but this is not the kind of question I am concerned with in this paper. It is my opinion that completeness is a desirable property from a language design point of view. Completeness for ML can be achieved by either *extending* the type system of ML to support polymorphic recursion, or *restricting* it by banning non-regular recursive datatypes. Strictly speaking, the first part of this claim is a conjecture, since the standard argument to show completeness [5] does not go through for this language.

1.3 Sketch of Incompleteness Proof

How can we show there is no ML-function that implements `fvars`?

Given a term of type `'a lam` it is clear that any implementation of `fvars` has to "look at" all `Lam` constructors in its argument. "Looking at" a constructor means to match it through pattern matching, i.e. each `Lam` is matched by a corresponding pattern. Notice that nested occurrences of `Lam` have different types. ML type checking guarantees that we only match values of type t against patterns of type t, so how does a well-typed implementation of `fvars` find patterns of all these types? The answer is: it can't. One can eliminate ML polymorphism by making enough syntactic copies of the polymorphic bits, but this technique is doomed to fail for proper polymorphic recursion where "enough" means: arbitrarily many. Technically, the proof proceeds by showing that a certain relation between environments and type environments which holds for all reachable environments forbids an implementation of `fvars`; it is the same relation one would typically use for a soundness proof.

There is a connection with the pumping-lemma of formal language theory: if we view the type `'a lam` as an infinite tree in the style of [1] then we can observe that this tree is not regular, it has infinitely many different subtrees. ML functions can only fully recognise values of regular types.

2 Operational Semantics

To make the claims precise we present here the operational semantics for a small functional programming language with recursion and pattern matching.

$$
\begin{array}{rl@{\qquad}rl}
e ::= & x \mid b \mid c \mid (\lambda m) \mid & & \\
& (e\, e') \mid (e, e') \mid & v ::= & b \mid c \mid c \cdot v \mid (v, v') \mid (E, m) \\
& \mathbf{let}\ x = e'\ \mathbf{in}\ e \mid & w ::= & v \mid \langle e \rangle \\
& \mathbf{fix}\ x = e'\ \mathbf{in}\ e & & \\
& & E ::= & \bullet \mid E[x \mapsto w] \\
m ::= & p.e \mid p.e; m & & \\
& & R ::= & E \mid \Box \\
p ::= & x \mid b \mid (c\, p) \mid (p, p') & &
\end{array}
$$

Table 1. Abstract Syntax

We consider expressions over the abstract syntax as shown in left-hand column of table 1, for which we assume the usual notational conventions for enriched λ-calculi, regarding the omission of parentheses etc. The metavariable e ranges over *expressions*, p over *patterns*, and m over *matches* (lists of pairs of patterns and expressions); x ranges over a set of variables and b and c over a set of constructors where b are thought to be nullary constructors. The set of variables is assumed to be countably infinite. In a match $p_1.e_1; \cdots; p_n.e_n$ the variables in p_i bind variables in e_i. Expressions of the form $\mathbf{fix}\, x = e'\, \mathbf{in}\, e$ model recursive bindings, i.e. x is recursively defined by e'.

This language is in similar form included in most functional programming languages. However, it is significantly more complex than the simple expression languages usually considered in related research papers, for example [7]. The reason for this complication lies in the issues we want to address which are inextricably linked with both pattern matching and recursion.

To define an operational semantics for this language we need a notion of value and environment, as defined in the right column of table 1. The metavariable v

ranges over *values* which are either constructors, constructor applications, pairs of values, or closures; w ranges over *environment entries* which are either values or expressions, the latter being a special way of representing fixpoints. E ranges over *environments*, finite lists of pairs of variables and environment entries; R ranges over pattern matching results which are either environments or $\Box$, where $\Box$ indicates that matching has failed.

We now can define the mentioned ternary evaluation relation $E \vdash e \to v$. We also define two 4-place relations: $E, v \vdash p \to R$ meaning that the value v matches the pattern p giving rise to the matching result R; $E, v \vdash m \to v'$ meaning that the function denoted by m (with free identifiers determined by E) evaluates to v' when applied to the argument v.

2.1 Pattern Matching and Function Application

The rules for pattern matching can be given independently from the others, see table 2. The rules for pattern matching are straightforward and follow essentially the rules for SML [6] with $\Box$ playing the rôle of FAIL. The only notable difference is that instead of returning the environment consisting of exactly the bindings caused by the pattern matching, we return the extension of the current environment with such bindings.

$$\frac{}{E, v \vdash x \to E[x \mapsto v]}$$

$$\frac{}{E, b \vdash b \to E} \qquad \frac{b \neq b'}{E, b' \vdash b \to \Box} \qquad \frac{}{E, c \cdot v \vdash b \to \Box}$$

$$\frac{E, v \vdash p \to E'}{E, c \cdot v \vdash (c\, p) \to E'} \qquad \frac{c \neq c'}{E, c' \cdot v \vdash (c\, p) \to \Box} \qquad \frac{}{E, b \vdash (c\, p) \to \Box}$$

$$\frac{E, v \vdash p \to E' \quad E', v' \vdash p' \to R}{E, (v, v') \vdash (p, p') \to R} \qquad \frac{E, v \vdash p \to \Box}{E, (v, v') \vdash (p, p') \to \Box}$$

Table 2. Pattern Matching

The rules for function application (table 3) and expression evaluation are mutually recursive. Function application is also defined very similar to [6], patterns are tried from left to right.

2.2 Expression Evaluation

The rules for expression evaluation in table 4 differ from SML's, because we have chosen a different and slightly more expressive method of unravelling recursion.

We have three rules for variable access: (i) we skip irrelevant environment entries, (ii) we access value entries, and (iii) we unravel recursive entries. The evaluation of constructors and λ-abstractions finishes instantly, as they are more

$$\frac{E, v \vdash p \to E' \quad E' \vdash e \to v'}{E, v \vdash p.e \to v'}$$

$$\frac{E, v \vdash p \to \Box \quad E, v \vdash m \to v'}{E, v \vdash p.e; m \to v'} \qquad \frac{E, v \vdash p \to E' \quad E' \vdash e \to v'}{E, v \vdash p.e; m \to v'}$$

Table 3. Function Application

$$\frac{x \neq x' \quad E \vdash x \to v}{E[x' \mapsto w] \vdash x \to v} \qquad \frac{}{E[x \mapsto v] \vdash x \to v} \qquad \frac{E[x \mapsto \langle e \rangle] \vdash e \to v}{E[x \mapsto \langle e \rangle] \vdash x \to v}$$

$$\frac{}{E \vdash b \to b} \qquad \frac{}{E \vdash c \to c} \qquad \frac{}{E \vdash (\lambda m) \to (E, m)}$$

$$\frac{E \vdash e \to v \quad E \vdash e' \to v'}{E \vdash (e, e') \to (v, v')} \qquad \frac{E \vdash e \to c \quad E \vdash e' \to v}{E \vdash (e\, e') \to c \cdot v}$$

$$\frac{E \vdash e \to (E', m) \quad E \vdash e' \to v \quad E', v \vdash m \to v'}{E \vdash (e\, e') \to v'}$$

$$\frac{E \vdash e' \to v \quad E[x \mapsto v] \vdash e \to v'}{E \vdash \mathbf{let}\; x = e' \;\mathbf{in}\; e \to v'} \qquad \frac{E[x \mapsto \langle e' \rangle] \vdash e \to v}{E \vdash \mathbf{fix}\; x = e' \;\mathbf{in}\; e \to v}$$

Table 4. Expression Evaluation

or less considered values themselves. The rules for pairs and constructor application are self-explanatory. Closure applications are defined in terms of the judgement form $E, v \vdash m \to r$. The rule for **let** does what one would expect; **fix** is similar, but the binding is recursive and the evaluation is delayed until the recursive identifier is used. So one can use **fix** to simulate lazy evaluation.

2.3 Properties

As already mentioned we can define a pointed transition system based on the operational semantics.

Definition 2.1. The pointed transition system M is defined as follows: environments are states, the empty environment $\bullet$ is the initial state, transitions are defined as:

$$E \stackrel{(e,x)}{\longrightarrow} E[x \mapsto v] \iff E \vdash e \to v$$

A transition of M can be viewed as evaluating a declaration in an ML interpreter, the states of M correspond closely to those states of an (untyped) ML interpreter the user can interact with.

Proposition 2.2. *M is deterministic; more specifically:*

1. *Let $E \vdash e \to v$ and $E \vdash e \to v'$. Then $v = v'$.*

2. Let $E, v \vdash m \to v'$ and $E, v \vdash m \to v''$. Then $v' = v''$.
3. Let $E, v \vdash p \to R$ and $E, v \vdash p \to R'$. Then $R = R'$.

Determinism simplifies the reasoning about programs significantly. In particular, bisimulation equivalence and trace equivalence coincide for deterministic transition systems; this means here that two environments are indistinguishable iff they evaluate the same sequences of expressions. One can sharpen that result: two environments are indistinguishable iff they evaluate the same expressions.

Since the bindings for program variables are purely syntactic, environments, values and expressions can be seen as terms over some many-sorted first-order signature and similarly the three judgement forms (plus the inequality of constructors and variables) as first-order predicate symbols. This allows the reading of our specification of the evaluation relation as a first-order logic program (a set of Horn clauses), which has exactly the desired meaning once we augment it with some symbols for the formation of constructors and program variables and the appropriate clauses for the inequality predicates for these sorts.

Consequences of this logic program are not only the judgements of the operational semantics but also certain open judgements, e.g. $E, b \vdash b \to E$ is not only derivable for any concrete E and b but also as an open judgement where E and b are meta-variables ranging over environments and constructors. Remark: we have notationally suppressed injections into coproducts. As always, derivable judgements are closed under substitutions (of meta-variables). For simplicity, we shall only consider meta-variables for values and not for other sorts.

Terms with meta-variables can be preordered by the subsumption preorder (see for instance [3]): $t \preceq u \iff \exists \theta.\, \widehat{\theta}(t) = u$. We write $\widehat{\theta}$ for the homomorphic extension of a substitution θ to a function on first-order terms.

Definition 2.3. Suppose J is a derivable judgement. A *principal judgement* for J is a derivable judgement J_0 such that $J_0 \preceq J$ and for all $J' \preceq J$ we have $J_0 \preceq J'$.

In other words, a principal judgement of J is an initial object in the preordered category of derivable judgements (with order $\preceq$) with terminal object J. We simply say that J is principal if it is a principal judgement of itself.

Proposition 2.4. *For any derivable judgement there is a principal judgement.*

One can show this proposition by constructing a principal judgement for a derivable judgement J: take its derivation tree, form the principal judgements of the premises and find the mgu for a unification problem that makes them fit the premises of the last applied rule. This is a well-defined construction because each derivable judgement (of this language) has a unique derivation tree.

Given an (open) judgement J, we say that a meta-variable x *occurs negatively* in J iff it occurs "left from $\vdash$", e.g. x occurs negatively in the application judgement $E, v \vdash m \to v'$ iff it occurs in either E or v or both.

Lemma 2.5. *Let J be a principal judgement. Then no meta-variable occurs more than once negatively in J.*

Proof. Sketch: by induction on the derivation tree, we simply replace shared by distinct meta-variables and obtain a more general derivable judgement. □

For lemma 2.5 it is vital that we have restricted ourself to meta-variables of value sort; meta-variables for constructors or program variables may well be shared negatively in principal judgements.

Lemma 2.6. *Let $J = J'[c \cdot v/x]$ be a principal judgement such that x occurs negatively in J'. Then the derivation tree of J contains a subtree with conclusion of the form $E, c \cdot v \vdash (c\,p) \rightarrow E'$.*

Proof. Sketch: the proof goes by induction on the height of the derivation tree of J. The argument is typically as in this example: let J be the judgement $E \vdash (e\,e') \rightarrow v'$. Unless e evaluates to some c' (a case we ignore here) it must have been derived from the premises $J_1 = E \vdash e \rightarrow (E', m)$, $J_2 = E \vdash e' \rightarrow v''$, and $J_3 = E', v'' \vdash m \rightarrow v'$ for some E', m, and v''. We can consider the principal judgements J_i' for the judgements J_i and observe that J is derived from the mgu of a system of equations (making the components of the J_i' fit to the schema of the closure-application rule) none of which mentioning the constructor c. This means that at least one of these principal judgements has a negative occurrence of some $c \cdot v_0$ (which is mapped by the mgu to $c \cdot v$): we can apply the induction hypothesis to the corresponding premise and obtain the desired result by applying the mgu to the derivation trees of the principal judgements. □

Lemma 2.6 says informally that when we need to read a constructor in order to get a result we have to do so by matching a constructor pattern against the value that has this constructor on top. This should hardly be surprising. The restriction to negative occurrences is necessary, because we can produce a constructor value in a positive occurrence by evaluating a constructor application.

Consider the values $v_n = \mathtt{Lam}^n(\mathtt{Var}(\mathtt{An}^n(v)))$ (we use superscript for iterating functions) and $v_n' = \mathtt{Lam}^n(\mathtt{Var}(\mathtt{An}^{n-1}(\mathtt{One})))$ of type `t lam` where `t` is the type of v. Clearly, v_n "contains a free variable" and v_n' does not. In other words:

Lemma 2.7. *Suppose (E, m) implements* `fvars`*.*

1. *The judgements $J = E, v_n \vdash m \rightarrow c \cdot (v, b)$ and $J' = E, v_n' \vdash m \rightarrow b$ are derivable, where b is* `[]` *and c the list constructor* `::`*.*
2. *The principal judgement of J' has the form $E_1, v_n' \vdash m \rightarrow b$.*

Proof. The first part is obvious as this is the required behaviour of `fvars`. Suppose the principal judgements of J and J' are $E_0, v_0'' \vdash m \rightarrow v_c$ and $E_1, v_1'' \vdash m \rightarrow v_b$, respectively.

By assumption there are substitutions σ_i mapping E_i to E, also $\widehat{\sigma_1}(v_1'') = v_n'$ which implies $v_1'' \preceq v_n'$. By lemma 2.5 the meta-variables in v_1'' and E_1 are disjoint. If $v_1'' \prec v_n'$ then we also have $v_1'' \prec v_n$ and by the previous observation there is a substitution v such that $\widehat{v}(v_1'') = v_n$ and $\widehat{v}(E_1) = E$. In particular, we can derive $E, v_n \vdash m \to \widehat{v}(v_b)$. By determinism and the fact we can derive J we must have $\widehat{v}(v_b) = c \cdot (v, b)$. We also have $\widehat{\sigma_1}(v_b) = b$. Since b and c are different constructors, v_b must be a meta-variable. By determinism and substitutivity of judgements v_b must occur in either E_1 or v_1''. It cannot occur in v_1'', because $b = \widehat{\sigma_1}(v_b)$ does not occur in v_n'; it cannot occur in E_1 either, because the substitutions v and σ_1 agree on E_1. Hence the assumption $v_1'' \prec v_n'$ leads to a contradiction and we must have $v_1'' = v_n'$ since v_n' is ground. □

Lemma 2.7 means that we can apply lemma 2.6 to *all* constructors in v_n', e.g. for each `Lam` there has to be a corresponding pattern match. Since this is true for the principal judgement it trivially follows for J' itself.

Corollary 2.8. *Consider the judgement J' of lemma 2.7 (for some given n). Its derivation tree contains pattern matching judgements of the form $E_k, \mathtt{Lam}^k(\mathtt{Var}(\mathtt{An}^{n-1}(\mathtt{One}))) \vdash (\mathtt{Lam}\, p_k) \to E_k'$ for all k, $0 \leq k \leq n$.*

Claim: For any occurrence of a pattern p (or: expression e', match m') in the derivation tree of $E \vdash e \to v$ (including v) there is a corresponding occurrence of p (or e', m', respectively) in either E or e; we make analogous claims for derivation trees of $E, v \vdash m \to v'$ and $E, v \vdash p \to E'$.

That any occurrence of a piece of syntax can be traced back to E or e is obvious from the shape of the rules: no rule uses "free" syntax in their premises. A pattern p may occur several times in E and/or e, but each occurrence of p in the derivation tree can be traced to a particular occurrence of p in either E or e, because no rule uses repeated patterns in its conclusion. One could make this tracing precise by giving patterns unique labels, similarly to labelling techniques for the λ-calculus used for the tracing of residuals of redexes.

2.4 let-expansion

The presence of **let**-expressions in the language only becomes significant once we type them in a special way; operationally, the expression $\mathbf{let}\ x = e\ \mathbf{in}\ e'$ is clearly equivalent to $(\lambda x.\, e')\, e$, but ML's typing rules differ. However, there is a different way of removing **let**-expressions that preserves well-typedness, it is expressed by the second-order rewrite rule $\mathbf{let}\, x = e\, \mathbf{in}\, Z(x) \;\Rightarrow\; Z(e)$. This rule replaces all free occurrences of x in e' by e.

The rule is only correct if we prevent the free program variables of e to be bound by the substitution. This is automatically taken care of if we use higher-order abstract syntax for the programming language such that its binding constructs (**let**, **fix**, and λ) are all expressed in terms of the binding construct of the rewrite language. Alternatively, we can stick to first-order terms and first-order rewriting and explicitly rename bound program variables whenever necessary. We leave the details to the imagination of the reader and use $\Rightarrow$ for the pre-congruence closure of this rewrite relation.

Proposition 2.9. *The relation $\Rightarrow$ (on environments and values) is a subrelation of $\precapprox$; also, $E \xrightarrow{(e,x)} E'$ and $e \Rightarrow e'$ imply $\exists E''.\, E \xrightarrow{(e',x)} E''$ and $E' \precapprox E''$.*

Proposition 2.9 can be proved most easily for a first-order presentation of $\Rightarrow$. The statement is only in terms of $\precapprox$ rather than $\approx$ since the expression e in $\mathbf{let}\ x = e\ \mathbf{in}\ e'$ needs to be evaluated for the evaluation of the whole expression but this might be avoided by the expansion, most notably if x does not occur at all in e'.

Proposition 2.10. *The relation $\Rightarrow$ is strongly normalising.*

As a consequence any environment has a normal form w.r.t. the relation $\Rightarrow$ which simulates it; the normal forms of $\Rightarrow$ do not contain **let**-expressions. Thus for any implementation (E, m) of `fvars` its **let**-free normal form (E', m') implements `fvars` as well.

3 Type System

We briefly repeat here the type system of SML [6] adapted to our little language. First we give an abstract syntax for types, type schemes and type environments. Types t are first-order terms over a signature with two distinguished binary function symbols "$\rightarrow$" and "$\times$". A type scheme σ has the form $\forall\alpha_1.\cdots.\alpha_k.t$ which we abbreviate as $\forall\boldsymbol{\alpha}_k.\,t$. Type environments B associate (program) variables with type schemes, the notation is analogous to environments.

Type schemes can be seen as polymorphic types. The $\forall$ sign in $\forall\alpha.\,\sigma$ is a variable binder, i.e. $\mathrm{FV}(\forall\alpha.\,\sigma) = \mathrm{FV}(\sigma) \setminus \{\alpha\}$, and for types: $\mathrm{FV}(\alpha) = \{\alpha\}$ and $\mathrm{FV}(f(t_1, \ldots, t_n)) = \bigcup_{1\le i\le n} \mathrm{FV}(t_i)$. Free variables for type environments are analogous to environments: $\mathrm{FV}(B[x \mapsto \sigma]) = \mathrm{FV}(\sigma) \cup \mathrm{FV}(B)$, $\mathrm{FV}(\bullet) = \emptyset$. We write $\sigma \triangleright t$ iff for some type variables $\alpha_1, \ldots, \alpha_k$ and some types $t', t_1, \ldots, t_k$ we have $\sigma = \forall\alpha_1.\cdots.\forall\alpha_k.\,t'$ and $t'[t_1/\alpha_1, \ldots, t_k/\alpha_k] = t$. We write $\mathrm{Clos}_B(t)$ for some type scheme $\forall\boldsymbol{\alpha}_k.\,t$ such that $\{\alpha_1, \ldots, \alpha_k\} = \mathrm{FV}(t) \setminus \mathrm{FV}(B)$.

In the following we will assume the existence of a function A that maps constructors to type schemes. Moreover, this function should have the following properties: $\forall b.\exists f, t_1, \ldots, t_k.\, A(b) = \forall\boldsymbol{\alpha}_n.\, f(t_1, \ldots, t_k) \wedge f \notin \{\times, \rightarrow\}$ which is the assumption that there are no nullary constructors for function and product types and similarly for non-nullary constructors: $\forall c.\, \exists f, t, t_1, \ldots, t_k.\, A(c) = \forall\boldsymbol{\alpha}_n.\,(t \rightarrow f(t_1, \ldots, t_k)) \wedge f \notin \{\times, \rightarrow\} \wedge \mathrm{FV}(t) \subseteq \bigcup_{1\le i\le k} \mathrm{FV}(t_i)$. The condition on $\mathrm{FV}(t)$ ensures that the type checking equivalent of pattern matching is deterministic, i.e. whenever $B, t \vdash p : B_1$ and $B, t \vdash p : B_2$ then $B_1 = B_2$. This property is also crucial for the soundness of pattern matching.

Similar to the presentation of evaluation, the type system is given in three parts, rules for type checking abstractions of the form $B \vdash m : t$, rules for type-checking patterns of the form $B, t \vdash p : B'$ and rules for type-checking expressions of the form $B \vdash e : t$.

As one would expect, the rule that crucially restricts the interaction between recursion and polymorphism is the typing rule for the **fix** construct. By replacing the premise $B[x \mapsto t'] \vdash e' : t'$ in that rule by the two premises $B[x \mapsto \sigma] \vdash e' : t'$ and $\mathrm{Clos}_B(t') = \sigma$ we could transform the type system into one with full-blown polymorphic recursion.

$$\frac{B,t \vdash p : B' \quad B' \vdash e : t'}{B \vdash p.e : t \to t'} \qquad \frac{B \vdash p.e : t \quad B \vdash m : t}{B \vdash p.e; m : t}$$

$$\frac{}{B,t \vdash x : B[x \mapsto t]} \qquad \frac{B,t \vdash p : B' \quad B',t' \vdash p' : B''}{B,t \times t' \vdash (p,p') : B''}$$

$$\frac{B \vdash b : t}{B,t \vdash b : B} \qquad \frac{B \vdash c : (t' \to t) \quad B,t' \vdash p : B'}{B,t \vdash (c\, p) : B'}$$

$$\frac{A(b) \rhd t}{B \vdash b : t} \qquad \frac{A(c) \rhd t}{B \vdash c : t} \qquad \frac{B \vdash m : t}{B \vdash \lambda m : t}$$

$$\frac{\sigma \rhd t}{B[x \mapsto \sigma] \vdash x : t} \qquad \frac{x \neq x' \quad B \vdash x : t}{B[x' \mapsto \sigma] \vdash x : t}$$

$$\frac{B \vdash e : (t' \to t) \quad B \vdash e' : t'}{B \vdash (e\, e') : t} \qquad \frac{B \vdash e : t \quad B \vdash e' : t'}{B \vdash (e, e') : t \times t'}$$

$$\frac{B \vdash e' : t' \quad \sigma = \mathrm{Clos}_B(t') \quad B[x \mapsto \sigma] \vdash e : t}{B \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : t}$$

$$\frac{B[x \mapsto t'] \vdash e' : t' \quad B[x \mapsto t'] \vdash e : t}{B \vdash \mathbf{fix}\ x = e'\ \mathbf{in}\ e : t}$$

Table 5. Type System

Definition 3.1. The pointed transition system T is defined as follows: type environments are states, the empty type environment $\bullet$ is the initial state, transitions are defined as:

$$B \xrightarrow{(e,x)} B[x \mapsto \sigma] \iff \exists t.\, B \vdash e : t \wedge \mathrm{Clos}_B(t) = \sigma$$

Although T is not deterministic (an expression can have many types), it is still the case that bisimulation and trace equivalence coincide and we also have a similar compression property as in the operational semantics, i.e. two type environments are equivalent iff they type-check the same expressions. More specifically, it is not difficult to see that two type environments are equivalent iff they have the same domain and are pointwise equivalent, where the equivalence of type schemes just means that they can instantiate to the same types. This equivalence is exactly the equality of type schemes and static environments in the static semantics of SML [6].

Proposition 2.9 carries over to the transition system T: judgements are substitutive and free syntactic type variables are just like meta-variables for types. The **let**-expansion then operates on types and type schemes like a β-reduction. As a consequence, the **let**-free normal form of any well-typed expression is well-typed (in the same type environment). Together with the original proposition 2.9 this means in particular that any well-typed implementation of `fvars` gives rise to a well-typed **let**-free implementation.

From M and T we can define the typed transition system $M : T$ determined by the relation $l : l' \iff l = l'$ between labels. The states and transitions of

$$\frac{v : \forall\alpha.\, t}{v : t[t'/\alpha]} \qquad \frac{v : \sigma}{v : \forall\alpha.\, \sigma}$$

$$\frac{v : t \quad v' : t'}{(v, v') : t \times t'} \qquad \frac{E : B \quad B \vdash m : t}{(E, m) : t}$$

$$\frac{\bullet \vdash b : t}{b : t} \qquad \frac{\bullet \vdash c : (t' \to t) \quad v : t'}{c \cdot v : t} \qquad \frac{\bullet \vdash c : t}{c : t}$$

$$\frac{}{\bullet : \bullet} \qquad \frac{v : \sigma \quad E : B}{E[x \mapsto v] : B[x \mapsto \sigma]} \qquad \frac{E : B \quad B[x \mapsto t] \vdash e : t}{E[x \mapsto \langle e \rangle] : B[x \mapsto t]}$$

Table 6. Types of Values and Environments

$M : T$ capture the states and evaluations of a typed ML interpreter: we only ever evaluate expressions which are well-typed in the current environment and the resulting value is asserted the type we obtained from this type-check. One can formalise and prove a subject reduction theorem for $M : T$ by defining a well-typedness predicate P for its states and showing that P is an invariant; the soundness of $M : T$ would additionally require that the initial state satisfies P.

There is another way of defining exactly the same transition system, simply by defining judgements like $(E, B) \vdash e \to (v, t)$ as the combination of $E \vdash e \to v$ and $B \vdash e : t$. By considering the various cases for e we can derive inference rules for this judgement form which are largely defined in terms of themselves. For example, for closure application we get the following derived rule:

$$\frac{(E, B) \vdash e \to ((E', m), t' \to t) \quad (E, B) \vdash e' \to (v', t') \quad E', v' \vdash m \to v}{(E, B) \vdash (e\, e') \to (v, t)}$$

In general, the application judgement $E', v' \vdash m \to v$ of this derived rule has no corresponding typing judgement. In order to get such a thing we need some assumptions about the relationship between E and B. Table 6 defines such a relation $E : B$, which intuitively means that E has "type" B. Notice that this typing judgement is decidable and consequently it is much more restrictive than a proper semantic notion of type inhabitance would be like. Claim: if $E \Rightarrow E'$ and $E : B$ then $E' : B$.

The relation $E : B$ is an invariant of the transition system $M : T$, i.e. states (E, B) satisfying $E : B$ have only transitions into other such states.

Proposition 3.2. *Let $E : B$ and $(E, B) \vdash e \to (v, t)$. Then $v : t$.*

Proof. Sketch: we make analogous claims for pattern matching and function application and prove the result by induction on the height of the derivation tree. The non-trivial cases are closure application and the unravelling of recursion.

Consider our derived rule for closure application: We can apply the induction hypothesis to the premise $(E, B) \vdash e \to ((E', m), t' \to t)$ which gives us $(E', m) : t' \to t$. By the rules for types of values this is the case iff there is a B' such $E' : B'$

and $B' \vdash m : t' \to t$. Thus we also have $(E', B'), (v', t') \vdash m \to (v, t)$. Moreover, $E' : B'$ as we have already seen and $v' : t'$ follows from the induction hypothesis of the second premise. Thus we can apply the induction hypothesis to this statement as well (its derivation height is equal to the height of $E', v' \vdash m \to v$) and get the result.

Unravelling recursion: Assume $E = E'[x \mapsto \langle e \rangle]$ with $(E, B) \vdash x \to (v, t)$ and $E : B$. The first condition means $B \vdash x : t$ and $E \vdash e \to v$; from the second condition we get $B = B'[x \mapsto t']$ and $B \vdash e : t'$. Thus $B'[x \mapsto t'] \vdash x : t$ which implies $t = t'$. Putting all of this together gives us $(E, B) \vdash e \to (v, t)$ and we can use the induction hypothesis. □

Corollary 3.3. *For any reachable state (E, B) in $M : T$ we have $E : B$.*

Proposition 3.2 can be regarded as a subject reduction theorem of the typed language and its corollary as a (part of a) soundness statement. The subject reduction proof proves a bit more than just the proposition: from $E : B$ and $(E, B) \vdash e \to (v, t)$ it constructs a derivation tree in which all evaluation (application, pattern matching) steps are of such a guarded form.

Consider any pattern match $(E_0, B_0), (v_0, t_0) \vdash p \to (E_1, B_1)$ in the derivation tree of $(E, B) \vdash e \to (v, t)$ such that B contains only types (no proper type schemes) and E and e are **let**-free. As we mentioned earlier, the pattern p can be traced back to either E or e. In the former case, the judgement $B_0, t_0 \vdash p : B_1$ occurs in the derivation tree of $E : B$, in the latter case in the derivation tree of $B \vdash e : t$. This follows from the way these trees are constructed in the subject reduction proof. The absence of polymorphic bindings is vital: whenever an identifier x is accessed then we can create a proof that $v : t$ (its value v has type t) from the proof that $E : B$; if B is monomorphic then this derivation occurs as a subderivation of $E : B$. The further absence of **let**-expressions is necessary to maintain the absence of type schemes for the full attributed derivation tree.

Theorem 3.4. *No reachable state in $M : T$ contains an implementation of* `fvars`.

Proof. Suppose (E, B) were reachable and contained such an implementation (E_0, m_0). We know $E : B$ from corollary 3.3. From proposition 2.10 we know that E has a **let**-free normal form $E \Rightarrow^* E'$ and it is easy to see that $E' : B$. By proposition 2.9 we have $E \precapprox E'$. In particular, if we have $E_0, v_n' \vdash m_0 \to b$ (v_n' and b as mentioned earlier) then we also have $E_0', v_n' \vdash m_0' \to b$ for the **let**-free normal form (E_0', m_0') with a corresponding typing judgement $B_0 \vdash m_0' : t \to t'$. From corollary 2.8 we know for any n that the derivation tree of $E_0', v_n' \vdash m_0' \to b$ contains judgements of the form $E_k, \mathtt{Lam}^k(\mathtt{Var}(\mathtt{An}^{n-1}(\mathtt{One}))) \vdash (\mathtt{Lam}\, p_k) \to E_k'$ (for any $0 \leq k \leq n$) each of which having a corresponding typing judgement $B_k, t_k \vdash (\mathtt{Lam}\, p_k) \to B_k'$.

If B_0 is monomorphic then we know that these typing judgements occur in the proofs of either $E_0 : B_0$ or $B_0 \vdash m_0' : t \to t'$. But this is impossible because each t_k is (must be) different and for sufficiently large n there simply are not enough types in these proofs. If B_0 is not monomorphic then we can transform

(B_0, m_0', E_0') into something equivalent but monomorphic by linearising (E_0', m_0') in its polymorphic program variables: if x is used in different type instances in the proofs of $E_0' : B_0$ and $B_0 \vdash m_0' : t \rightarrow t'$ then we introduce a fresh program variable $x_{t''}$ for every type instance t'' of x's type scheme that occurs in these proofs and replace each occurrence of x in m_0' by the corresponding $x_{t''}$; we bind $x_{t''}$to t'' and the old value of x, respectively. Notice that thunks $\langle e \rangle$ are never polymorphic, i.e. x must be bound to some v (never $\langle e \rangle$) in E_0' if it is bound to a proper type scheme in B_0. □

Corollary 3.5. *$M : T$ is incomplete.*

4 Final Remarks

We have shown that ML's type system is incomplete: there are values that can be soundly given a type t which cannot be expressed in ML, even modulo observational equivalence. One can show with a standard argument (see [5]) that ML's type system is complete for datatypes with a regular tree structure. Considering the extended (undecidable) type system of [4], it seems likely that it is complete as well but the mentioned standard argument does not apply here.

Acknowledgements

Thanks to Claudio Russo for carefully reading an earlier version of this paper. Also thanks to the PLILP referees for some helpful suggestions.

The research reported here was supported by SERC grant GR/J07303.

References

1. F. Cardone and M. Coppo. Two extensions of Curry's type inference system. In P. Odifreddi, editor, *Logic and Computer Science*, pages 19–76. Academic Press, 1990.
2. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
3. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 244–320. Elsevier, 1990.
4. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
5. Stefan Kahrs. About the completeness of type systems. In *Proceedings ESSLLI Workshop on Observational Equivalence and Logical Equivalence*, 1996. (to appear).
6. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
7. Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988. CST-52-88.