An ASM Dynamic Semantics for Standard ML

Steven C. Cater and James K. Huggins

Kettering University
Computer Science Program
1700 W. Third Avenue
Flint, MI 48504-4898 USA
{scater,jhuggins}@kettering.edu

Abstract. Abstract State Machines (ASM) are a methodology for formally specifying computing systems. We use the ASM methodology to give the dynamic semantics of the functional programming language Standard ML. We give an operational semantics for Standard ML by means of an interpreter for Standard ML programs.

1 Introduction

The Abstract State Machines (ASM) methodology (Gurevich 1995) is a methodology for formally specifying computing systems (software, hardware, or mixed). First introduced in (Gurevich 1988) (under their former name, "evolving algebras") the ASM methodology is mathematically precise, yet general enough to be applicable to a wide variety of problem domains (Börger and Huggins 1998, Glässer 1999, Huggins 1999). The ASM Thesis (Gurevich 1999) asserts that any computing system can be described at its natural level of abstraction by an appropriate ASM.

Standard ML (SML) is a functional programming language. It has been described as a "safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules." (Lucent 1999)

In this paper, we describe the dynamic semantics of SML using the ASM methodology, using (Milner *et. al.* 1997) as our definition of SML. We describe the dynamic semantics of SML by describing an ASM which acts as an SML interpreter, executing an (appropriately pre-processed) SML program. This provides an operational semantics for SML; the effect of a given SML instruction can be seen in terms of the corresponding actions performed by the ASM.

We separate concerns in this work and restrict our attention to the dynamic semantics of SML, assuming that all static semantic analysis and checking has been performed in a pre-processing stage. We also follow (Milner *et. al.* 1997) in restricting our attention to the so-called "bare" language of SML: a subset of SML from which all other language constructs in SML can be derived. Notice that this does not minimize the scope of the work; using appropriate substitutions, any SML program can be translated into the bare language and thus given semantics by our ASM descriptions.

2 Sequential ASMs

Sequential ASMs (under their former name, evolving algebras) are introduced in (Gurevich 1993); a more detailed description of ASMs (including distributed characteristics) is given in (Gurevich 1995). We present here only those features of sequential ASMs necessary to understand this paper. Those already familiar with ASMs may wish to skip ahead to the next section.

2.1 States

The states of an ASM are first-order logical structures (*i.e.*, sets with functions defined on those sets), except that relations are treated as Boolean-valued functions. We expand on that definition below.

A *vocabulary* is a finite collection of function names, each with a fixed arity. Every ASM vocabulary contains the following *logic symbols*: nullary function names *true*, *false*, *undef*, the equality sign, (the names of) the usual Boolean operations, and a unary function name *Bool*. Some function symbols (such as *Bool*) are tagged as *relations*.

A *state* S of vocabulary Υ is a non-empty set X (the *superuniverse* of S), together with interpretations of all function symbols in Υ over X (the *basic functions* of S). A function symbol f of arity r is interpreted as an r-ary operation over X; if $r = 0$, f is interpreted as an element of X. (If $r = 1$, we often write $a.f$ instead of $f(a)$.) The interpretations of the function symbols *true*, *false*, and *undef* are distinct, and are operated upon by the Boolean operations in the usual way. (We often use the value *undef* to code partial functions as total functions whose value is *undef* outside the indicated domain.)

Let f be a relation symbol of arity r. We require that (the interpretation of) f is *true* or *false* for every r-tuple of elements of S. If f is unary, it can be viewed as a *universe*: the set of elements a for which $f(a)$ evaluates to *true*. For example, *Bool* is a universe consisting of the two elements (named) *true* and *false*.

Let f be an r-ary basic function and $U_0, \ldots, U_r$ be universes. We say that f has *type* $U_1 \times \ldots \times U_r \rightarrow U_0$ in a given state if $f(\bar{x})$ is in the universe U_0 for every $\bar{x} \in U_1 \times \ldots \times U_r$, and $f(\bar{x})$ has the value *undef* otherwise.

2.2 Updates

The simplest change that can occur to a state is the change of an interpretation of a function at one particular tuple of arguments. We formalize this notion.

A *location* of a state S is a pair $\ell = (f, \bar{x})$, where f is an r-ary function name in the vocabulary of S and $\bar{x}$ is an r-tuple of elements of (the superuniverse of) S. (If f is nullary, ℓ is simply f.) An *update* α of a state S is a pair (ℓ, y), where ℓ is a location of S and y is an element of S. To *fire* α at S, put y into location ℓ; that is, if $\ell = (f, \bar{x})$, redefine S to interpret $f(\bar{x})$ as y and leave everything else unchanged.

2.3 Transition Rules

We introduce rules for describing changes to states. At a given state S whose vocabulary includes that of a rule R, R gives rise to a set of updates; to execute R at S, fire all the updates in the corresponding update set. We suppose throughout that a state of discourse S has a sufficiently rich vocabulary.

An *update rule* R has the form

$$f(t_1, t_2, \ldots, t_n) := t_0$$

where f is an n-ary function name and each t_i is a term. (If $r = 0$, we write $f := t_0$ rather than $f() := t_0$.) The update set for R contains a single update (ℓ, y), where y is the value $(t_0)_S$ of t_0 at S, and $\ell = (f, (x_1, \ldots, x_r))$, where $x_i = (t_i)_S$. In other words, to execute R at S, set $f(x_1, \ldots, x_n)$ to y, where x_i is the value of t_i at S and y is the value of t_0 at S.

A *block rule* R is a sequence $R_1, \ldots, R_n$ of transition rules. (We write rules in such a sequence on separate lines or separated by commas on the same line.) To execute R at S, execute all the R_i at S simultaneously. That is, the update set of R at S is the union of the update sets of the R_i at S. If two or more of these updates are contradictory (that is, update the same location to different values), the update set is defined to be empty.

A *conditional rule* R has the form

$$\textbf{if } g \textbf{ then } R_0 \textbf{ else } R_1 \textbf{ endif}$$

where g (the *guard*) is a term and R_0, R_1 are rules. The meaning of R is the obvious one: if g evaluates to *true* in S, then the update set for R at S is the same as that for R_0 at S; otherwise, the update set for R at S is the same as that for R_1 at S.

A *universe extension rule* R has the form

$$\begin{array}{l}\textbf{extend } U \textbf{ with } v\\ \quad R_0(v)\\ \textbf{endextend}\end{array}$$

where U is a universe name, v is a variable, and $R_0(v)$ is a rule with free variable v. To execute R in state S, add to universe U a new element a not previously in any universe and execute rule R_0 in state S, with v bound to a.

2.4 Programs and Runs

A *program* Π is simply a transition rule (typically a block rule).

A *sequential run* ρ of program Π from an initial state S_0 is a sequence of states $S_0, S_1, \ldots$, where each S_{i+1} is obtained from S_i by executing program Π in state S_i.

A sequential ASM is thus given by a program and a collection of initial states; this determines a corresponding collection of runs of the ASM.

3 ASM for SML: Simple Expressions

We now describe an ASM which acts as an intepreter for SML programs. Since we are solely concerned with the dynamic semantics of SML rather than the static semantics, we assume that all static analysis has already been performed on the program to be interpreted. The input program will thus be represented in an abstract form (to be described). The ASM operates by walking through this form of the program, performing the required calculations and changes to the system state.

In this section, we focus on the evaluation of arbitrary SML expressions written in the "bare" SML language. In succeeding sections we expand our focus to include other features of SML.

3.1 Some Common Universes, Functions, and Abbreviations

An SML program is a collection of terms; consequently, we define a universe of *Terms* which is used to represent a given program. The inter-relationship between various terms in a program are described by various unary functions, most of which are described as they are needed. For example, we use the unary function *Next: Terms* $\rightarrow$ *Terms* to indicate the next term in a sequence of terms being evaluated linearly. Similarly, we use the unary function *Parent: Terms* $\rightarrow$ *Terms* to indicate the smallest enclosing term of a given term. For example, (the term representing) the expression "1+2" is the parent of (the term representing) the expression "1".

The nullary function *CurTerm: Term* is used to indicate the current term being evaluated (similar to the role of an instruction counter). *CurTerm* acts as the focus of attention for the ASM; the ASM at each step examines various information regarding *CurTerm* and makes the appropriate changes to the state.

To identify particular terms, we define a universe of *Kinds* and a unary function *Kind: Terms* $\rightarrow$ *Kinds* used to label individual terms. The members of *Kinds* will be specified as we proceed. For example, a term t representing a function application satisfies *Kind(t) = functionApp*, where *functionApp* is a nullary function indicating a unique element of *Kinds.* (A more intuitive name for *Kinds* might be "*Types*"; we choose not to use that name to avoid confusion with SML's extensive type system.)

The result of evaluating an SML expression is a value; consequently, we define a universe of *Values* comprising the set of possible results of expression evaluation. We use a unary function *Value: Terms* $\rightarrow$ *Values* to record the results of evaluating a particular term during the course of a computation; the ASM updates *Value* throughout the computation as values propagate from child terms to parent terms.

SML uses special values called exceptions to indicate the presence of errors during a computation; the existence of an exception alters the normal flow of control. We use a universe *Exceptions* (a subset of *Values*) to represent the set of these special values; a nullary function *CurException: Exceptions* indicates the current exception (value) which is propagating through the system. If no exception is propagating, *CurException* has the special value *undef*.

We use two simple abbreviations, shown in Fig. 1. *OK* is a term indicating that the computation in process is proceeding normally (*i.e.*, no exception is present). *KEEPGOING* is a rule used when the current term has been processed and attention should be passed to the next term in the sequence of terms. These abbreviations are introduced to improve the readability of the forthcoming rules; additionally, we will later re-define the KEEPGOING abbreviation as the need for a more complicated "keep-going" command becomes necessary.

OK ::= (CurException = undef)
KEEPGOING ::= (CurTerm := CurTerm.Next)

Fig. 1. Definition of OK and KEEPGOING abbreviations.

3.2 Constant Expressions

Evaluating a constant requires almost no work; the result of evaluating a constant is the corresponding constant value. We use a static function *ConstValue: Terms* $\rightarrow$ *Values* to map a constant expression to its corresponding value. The corresponding ASM rule, shown in Fig. 2, is straightforward.

---

if *OK* **and** *CurTerm.Kind = constant* **then**
    CurTerm.Value := CurTerm.ConstValue
    KEEPGOING
endif

Fig. 2. Evaluating constants.

3.3 Identifiers

In SML, all expressions are evaluated with respect to an *environment*: a finite collection of name bindings. Evaluating an identifier in SML results in the corresponding value stored in the current environment. Thus, to give the rules for identifier evaluation, we need to define how environments are represented in our ASM.

We use universes of *Envs* (for environments) and *EnvEntries* (for entries in an environment) to represent a given environment, and An environment contains a finite number of entries, indexed by identifier names; we use a universe of *Identifiers* to represent these names. The binary function *Lookup: Envs* $\times$ *Identifiers* $\rightarrow$ *EnvEntries* returns the environment entry corresponding to the specified identifier in the specified environment. The nullary function *CurEnv: Envs* indicates the environment currently being used for evaluating expressions.

Environment entries are not simply values; each entry also has an additional datum indicating whether the value corresponds to a value variable, a value constructor, or an exception constructor. We use a universe of *ValueTags* to indicate these kinds of data in an environment, along with unary (projection) functions *ValueEntry: EnvEntries* $\rightarrow$ *Values* and *TagEntry: EnvEntries* $\rightarrow$ *ValueTags* to extract the needed information from a given environment entry.

Having described how environments work in SML, the rule for evaluating an identifier simply accesses the identifier (given by a simple function *Identifier: Terms* $\rightarrow$ *Identifiers*) and extracts the corresponding value from the current environment. The rule is shown in Fig. 3.

---

if *OK and CurTerm.Kind = identifier* **then**
    CurTerm.Value := Lookup(CurTerm.Identifier, CurEnv).ValueEntry
    KEEPGOING
endif

Fig. 3. Identifiers.

3.4 Records

In SML, a record is represented as a sequence of label-expression pairs; the result of evaluating a record is a finite mapping of the labels in the sequence to the corresponding values (as evaluated in the current environment).

Thus, we need a universe of *Maps* corresponding to these finite pairs of labels and expressions. (Note that *Maps* is not the same universe as *Envs*; environments carry additional information which record maps do not.) We use functions *CreateMap: Identifiers* $\times$ *Values* $\rightarrow$ *Maps* and *AddToMap: Maps* $\times$ *Identifiers* $\times$ *Values* $\rightarrow$ *Maps* to construct these maps.

Record expressions are represented in our ASM as a labeled expression optionally followed by a record expression (following the natural recursive definition of a sequence). Our ASM rules for evaluating records (shown in Fig. 4) move through this sequence of record expressions, constructing the corresponding map. (The rule below uses functions *Expr: Terms* $\rightarrow$ *Terms* and *NextRecord: Terms* $\rightarrow$ *Terms* to indicate the two components of a record expression; we implicitly introduce component functions such as these in future rules without comment.)

```
if OK and CurTerm.Kind = recordExpr then
    if CurTerm.Expr.Value = undef then CurTerm := CurTerm.Expr
    elseif CurTerm.NextRecord = undef then
        CurTerm.Value := CreateMap(CurTerm.Label, CurTerm.Expr.Value)
        CurTerm.Expr.Value := undef
        KEEPGOING
    elseif CurTerm.NextRecord.Value = undef then
        CurTerm := CurTerm.NextRecord
    else
        CurTerm.Value := AddToMap(CurTerm.NextRecord.Value,
                                  CurTerm.Label, CurTerm.Expr.Value)
        CurTerm.Expr.Value := undef, CurTerm.NextRecord.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 4. Records.

3.5 Reference Terms

In SML, references are pointers to storage locations maintained in system memory. References are used to store and retrieve values outside of the usual environment and expression evaluation mechanism; they thus break the "pure" functional programming paradigm.

Evaluating reference term of the form "`ref t`" causes a new memory location to be allocated and returned as the value of the expression. Additionally, the new memory location is initialized to the value of the argument expression `t`, as evaluated in the current environment.

Thus, we need a new universe of *Addresses* to represent references, and a unary function *Store: Addresses* $\rightarrow$ *values* to represent the current memory store. Evaluating a reference term thus requires allocating a new memory address and initializing the store appropriately. The resulting rule is shown in Fig. 5.

3.6 Assignment

Assignment terms change the values stored in system memory. Syntactically, an assignment appears in the "bare" language as the application of the special function ":=" to a single value: a record in which the label "1" is bound to the memory address to be changed and the label "2" is bound to the new value.

Recall that the value of a record is a map. In order to extract the necessary values from the argument record, we use a binary function *MapLookup: Maps* $\times$ *Identifiers* $\rightarrow$ *Values*. The rule (shown in Fig. 6) extracts the necessary information and performs the assignment.

All expressions in SML evaluate to a value; however, assignment expressions are evaluated for their side-effects rather than any value they might return. SML

```
if OK and CurTerm.Kind = reference then
    if CurTerm.Argument.Value = undef then CurTerm := CurTerm.Argument
    else
        extend Addresses with a
            CurTerm.Value := a, Store(a) := CurTerm.Argument.Value
        endextend
        CurTerm.Argument.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 5. Reference terms.

uses the special value "unit" to represent the value of an assignment expression; we use a corresponding distinguished element *Unit: Values*.

```
if OK and CurTerm.Kind = assign then
    if CurTerm.Argument.Value = undef then CurTerm := CurTerm.Argument
    else
        Store(MapLookup(CurTerm.Argument.Value,"1")) :=
            MapLookup(CurTerm.Argument.Value,"2"))
        CurTerm.Value := Unit, CurTerm.Argument.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 6. Assignment expressions.

3.7 Raising and Propagating Exceptions

A "`raise`" expression takes an argument which evaluates to an exception value. Evaluating such an expression causes the argument to be evaluated; the resulting value is then set to propagate through the system (in our ASMs, this is performed by assigning the value to *CurException*). The rule for evaluating such expressions is shown in Fig. 7.

Note that this rule sets *CurException* to a value other than *undef*, which falsifies the *OK* abbreviation.

Exceptions propagate by moving from child term to parent term repeatedly until a `handle` expression is found. The rule shown in Fig. 8 performs this activity. Note that the guard for this rule differs from the guards used in previous rules; the presence of a propagating exception makes the other rules inapplicable in this situation.

if *OK and CurTerm.Kind = raise* **then**
  if *CurTerm.Argument.Value = undef* **then** *CurTerm := CurTerm.Argument*
  else
    CurException := CurTerm.Argument.Value
    CurTerm.Argument.Value := undef, CurTerm := CurTerm.Parent
  endif
endif

Fig. 7. Raise expressions.

if *CurException* $\neq$ *undef* **and** *CurTerm.Kind* $\neq$ *handle* **then**
  CurTerm.Value := undef, CurTerm := CurTerm.Parent
endif

Fig. 8. Propagating exceptions.

3.8 Structure Restrictions

In SML, a structure restriction expression contains both an expression corresponding to a structure value and a signature term. Signature terms mainly play a role in the static analysis of SML; however, a structure restriction term evaluates to the structure value of the expression, with only those elements named in the signature visible. We model this behavior with a universe of *signatures* and a function *Restrict: Values* $\times$ *Signatures* $\rightarrow$ *Values*. The ASM rule is shown in Fig. 9.

if *OK* **and** *CurTerm.Kind = structureRestriction* **then**
  if *CurTerm.Structure.Value = undef* **then** *CurTerm := CurTerm.Structure*
  else
    CurTerm.Value := Restrict(CurTerm.Structure.Value, CurTerm.Signature)
    CurTerm.Structure.Value := undef
    KEEPGOING
  endif
endif

Fig. 9. Structure restrictions.

4 ASM for SML: Function Applications

In this section, we focus on the evaluation of SML expressions involving function application, which involve evaluating expressions in environments other than the current environment. We also discuss other SML expressions whose semantics involve changing the current evaluation environment.

4.1 Function Closures

A function closure is a match rule (in brief, a pattern-matching rule for evaluating the function) along with the environment to be used in evaluating that match rule. Function closures are created by evaluating statements of the form "`fn` *match*", where *match* is a match rule. The environment currently in use is bound to the specified match rule for later use.

We consequently make use of a universe of *FunctionClosures* representing this information, with construction function *MakeFunction: Terms* $\times$ *Envs* $\rightarrow$ *FunctionClosures* and projection functions *MatchBinding: FunctionClosures* $\rightarrow$ *Terms* and *EnvBinding: FunctionClosures* $\rightarrow$ *Envs*. The rule for evaluating function closure expressions is shown in Fig. 10.

---

if *OK* **and** *CurTerm.Kind* = *functionClosure* **then**
    CurTerm.Value := *MakeFunction* (*CurTerm.MatchTerm*, *CurEnv*)
    KEEPGOING
endif

Fig. 10. Function closures.

4.2 Function Applications: Preliminaries and Special Cases

Several forms of function application are present in SML. In all of these cases, a function application consists of two terms; a term corresponding to the function to be applied, and a term corresponding to the argument for that function. (All functions in the "bare" language are unary; functions of higher arity are simulated by records.) The rule shown in Fig. 11 performs this evaluation in all cases.

---

if *OK* **and** *CurTerm.Kind* = *functionApp* **then**
    if *CurTerm.LeftExpr.Value* = *undef* **then** *CurTerm* := *CurTerm.LeftExpr*
    elseif *CurTerm.RightExpr.Value* = *undef* **then** *CurTerm* := *CurTerm.RightExpr*
    endif
endif

Fig. 11. Evaluating arguments for function application.

One special case occurs when the function expression is not a function at all, but a value constructor (*i.e.*, an identifier). In this case, the value of the expression is a pair consisting of the constructor and the argument value. We use the function *Pair: Values* $\times$ *Values* $\rightarrow$ *Values* to construct such pairs. The rule for constructor application is shown in Fig. 12.

---

if *OK* **and** *CurTerm.Kind = functionApp* **and** *CurTerm.RightExpr.Value* $\neq$ *undef*
then
    if *CurTerm.LeftExpr.Value.Identifier* **then**
        CurTerm.Value := Pair(CurTerm.LeftExpr.Value, CurTerm.RightExpr.Value)
        CurTerm.LeftExpr.Value := undef, CurTerm.RightExpr.Value := undef
        KEEPGOING
    endif
endif

Fig. 12. Constructor application.

A second special case occurs when the function to be applied is a built-in operator. We use a universe *PrimitiveOps* to represent such primitive operations, as well as a function *Apply: PrimitiveOps* $\times$ *Values* $\rightarrow$ *Values* to represent the definition of such primitive operators. Thus, our rule simply uses the *Apply* function to generate the appropriate value.

Applying a primitive operator may also generate an exception; thus, if the value returned by *Apply* is an exception, we need to initiate exception handling (as in the case of a `raise` expression). Thus, the rule shown in Fig. 13 checks the resulting value before passing it along.

---

if *OK* **and** *CurTerm.Kind = functionApp* **and** *CurTerm.RightExpr.Value* $\neq$ *undef*
then
    if *CurTerm.LeftExpr.Value.PrimitiveOp* **then**
        let *Result = Apply(CurTerm.LeftExpr.Value, CurTerm.RightExpr.Value)* **in**
            if *Result.Exception* **then** *CurException := Result*
            else
                CurTerm.Value := Result
                KEEPGOING
            endif
        endlet
        CurTerm.LeftExpr.Value := undef, CurTerm.RightExpr.Value := undef
    endif
endif

Fig. 13. Primitive operator application.

4.3 Function Application to Function Closures

Here we handle the case of applying a user-defined function closure to an argument term. As seen above, function closures include a match expression and a current environment. Evaluating such a function application involves evaluating the specified match expression against the value of the function argument; however, such evaluation occurs not in the current environment, but in the environment specified in the function closure. (We defer our discussion of evaluating match terms against values.)

Of course, evaluating this match term in the new environment could involve evaluating other function applications, requiring evaluating other match terms in other environments, and so on. It becomes clear that we need a stack-like structure to use in evaluating terms in environments different from the current environment, while still maintaining the current environment for future use.

The elements which need to be stored in an activation record upon our stack are the following: the intermediate expression values already generated and stored in the function *Value*, the current focus of attention stored in *CurTerm*, the current evaluation environment stored in *CurEnv*, and the current exception propagating stored in *CurException*. Rather than creating an explicit stack structure (and following the precedent in (Gurevich and Huggins 1993)), we use the universe *Naturals* of natural numbers and redefine the functions named above to have the following signatures:

$$\begin{array}{l} \textit{Value: Terms} \times \textit{Naturals} \rightarrow \textit{Values} \\ \textit{CurTerm: Naturals} \rightarrow \textit{Terms} \\ \textit{CurEnv: Naturals} \rightarrow \textit{Envs} \\ \textit{CurException: Naturals} \rightarrow \textit{Exceptions} \end{array}$$

The extra numerical argument to each of these functions is used to indicate the level of the stack at which the given value is stored. For example, *CurTerm(*3*)* indicates the term currently being evaluated at level 3 of the stack. A new distinguished function *StackLevel: Naturals* indicates the current position on the stack; initially, *StackLevel = 0*. Thus, *CurTerm(StackLevel)* indicates the term currently being evaluated.

This requires changes to all the rules previously presented; to simplify our presentation (and following the precedent in (Börger and Rosenzweig 1994)), we abbreviate *CurTerm(StackLevel)* by *CurTerm*, *Value(X,StackLevel)* by *Value(X)*, and so on. In any function hereafter with an argument belonging to *Naturals*, we suppress that argument if its value is *StackLevel*; we explicitly specify the argument if it is a term other than *StackLevel* or if clarity would be better served by explicit specification. This set of abbreviations allows us to ignore the stack in situations where its presence is semantically uninteresting, and also allows us to use the previous rules without modification (other than, of course, the above abbreviations).

We introduce some functions used in the process of making a context-switch dealing with the call stack. *Target: Naturals → Terms* indicates the top-level term which is to be evaluated at this level of the stack. *CurMatchValue: Naturals*

→ *Values* indicates the value which is to be matched against *Target* as part of this evaluation process. *ReturnValue: Values* is used to store the return value when a given term evaluation on the stack completes.

We often make use of the *EVALUATE* abbreviation, shown in Fig. 14. The idea is that *EVALUATE(t, e, v)* attempts to match term t against value v using environment e. The abbreviation works by implicitly creating the next entry on the call stack and transferring control to term t. When that term completes its evaluation, *ReturnValue* will have the desired result and *StackLevel* will be reset to the proper value. The alternate form *EVALUATE(t,e)* is used in situations where a match value is not required.

```
EVALUATE(term, env, matchval) ::=
    Target(StackLevel + 1) := term
    CurTerm(StackLevel + 1) := term
    CurEnv(StackLevel + 1) := env
    CurMatchVal(StackLevel + 1) := matchval
    StackLevel := StackLevel + 1

EVALUATE(term, env) ::= EVALUATE(term, env, undef)
```

Fig. 14. *EVALUATE* abbreviation.

Finally, we can present the rule for performing a function application to a function closure. The rule shown in Fig. 15 simply invokes the specified function closure, attempting to match the specified argument against the function closure term.

```
if OK and CurTerm.Kind = functionApp and CurTerm.RightExpr.Value ≠ undef
      and CurTerm.LeftExpr.Value.FunctionClosure then
   if ReturnValue = undef then
      EVALUATE(CurTerm.LeftExpr.Value.MatchBinding,
                    CurTerm.LeftExpr.Value.EnvBinding,
                    CurTerm.RightExpr.Value)
   else
      CurTerm.Value := ReturnValue, ReturnValue := undef
      CurTerm.LeftExpr.Value := undef, CurTerm.RightExpr.Value := undef
      KEEPGOING
   endif
endif
```

Fig. 15. Function application to function closures.

4.4 Returning from Function Closures

The rules presented in the last section provide only half of what is needed for handling calls to function closures. The rules show how to invoke the evaluation of another expression in another environment; they fail to show what to do when evaluation of that expression has been completed. Here we complete the picture.

We begin by re-defining the abbreviation *KEEPGOING*. Recall that KEEPGOING works by moving *CurTerm* to the next expression to be evaluated in the program. This behavior should be suspended when *CurTerm* has returned to *Target*; at this point, the value stored at *Target* should be returned to the caller. The first half of this process is shown in the new definition of *KEEPGOING* in Fig. 16; it transfers control in such situations to a new distinguised element *ReturnTerm: Terms*, where we perform the actual return.

```
KEEPGOING ::=
    if CurTerm = Target then CurTerm := ReturnTerm
    else CurTerm := CurTerm.Next
    endif
```

Fig. 16. New definition of the *KEEPGOING* abbreviation.

We now show what happens when control reaches *ReturnTerm*. Here we simply have to keep the promises we made earlier: place the proper return value into *ReturnValue* and re-establish the previous evaluation context. Note that we have to deal with situations in which an exception is propagating as well; in such situations, we return the exception in both *ReturnValue* and *CurException*. The rule is shown in Fig. 17.

```
if CurTerm = ReturnTerm then
    if CurException = undef then ReturnValue := Value(Target)
    else
        ReturnValue := CurException, CurException(StackValue-1) := CurException
    endif
    Target.Value := undef, CurException := undef
    StackLevel := StackLevel - 1
endif
```

Fig. 17. Returning from a function call.

4.5 Handle Expressions

A handle expression is used to encapsulate exception processing over a given term. The argument term is evaluated; if no exception occurs while evaluating the argument, the handle expression does nothing and control continues on as usual. The rule for this case is shown in Fig. 18; notice that the truth of the guard OK indicates that no exceptions have occurred while processing the argument term.

---

if *OK and CurTerm.Kind = handle* **then**
  if *CurTerm.Expr.Value = undef* **then** *CurTerm := CurTerm.Expr*
  else
    CurTerm.Value := CurTerm.Expr.Value, CurTerm.Expr.Value := undef
    KEEPGOING
  endif
endif

Fig. 18. Normal processing of handle expressions.

Associated with each handle expression is a match rule (similar to that used in function definitions). If a propagating exception reaches a handle expression, we attempt to match the propagating exception (which is, after all, a value) against the associated match rule, as if we made a function call to the match rule. If the match rule returns a value or an exception, we allow that value or exception to continue propagating. The special value *Fail: Values* may be returned if the match rule fails to generate a result; in such situations, we allow the old exception to continue propagating. We use our context-switch mechanism introduced in the last section to perform this evaluation. The corresponding rule is shown in Fig. 19.

4.6 Let Expressions

Let expressions contain a set of bindings (which generate an environment) and an expression which should be evaluated with respect to those bindings and the current environment. Those bindings should only be added to the current environment to evaluate the target expression; afterwards, the current environment should revert its previous value.

We can use the same context-switch mechanism introduced previously to provide semantics for let expressions. We first evaluate the set of attached bindings, which generates an environment. We then combine that environment with the current environment, using a function *CombineEnv: Envs* $\times$ *Envs* $\rightarrow$ *Envs* to perform the combination appropriately. The new combined environment is then used as the basis for a context-switch to evaluate the target expression; thus, the new bindings implicitly disappear (as desired) when the call returns. The rule for evaluating let expressions is shown in Fig. 20.

```
if CurException ≠ undef and CurTerm.Kind = handle then
    if ReturnValue = undef then
        EVALUATE(CurTerm.Match, CurEnv, CurException)
    elseif ReturnValue = Fail or ReturnValue.Exception then
        CurTerm.Value := undef, CurTerm := CurTerm.Parent
    else
        CurTerm.Value := ReturnValue, CurException := undef
        KEEPGOING
    endif
    ReturnValue := undef
endif
```

Fig. 19. Exception processing of handle expressions.

```
if OK and CurTerm.Kind = let then
    if CurTerm.LeftExpr.Value = undef then CurTerm := CurTerm.LeftExpr
    elseif ReturnValue = undef then
        EVALUATE(CurTerm.RightExpr,
                     CombineEnv(CurEnv, CurTerm.LeftExpr.Value))
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        CurTerm.LeftExpr.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 20. Let expressions.

4.7 Functor Applications

A functor in SML is a quadruple *(id, sig, exp, env)*. Applying a value to a functor results in the evaluation of expression *exp* in environment *env*, where the identifier *id* is bound to the value of the functor argument in the current environment. The argument is restricted by signature *sig* before the evaluation takes place.

We use functions *FunctID: Values → Identifiers*, *FunctSig: Values → Signatures*, *FunctExpr: Values → Terms*, and *FunctEnv: Values → Envs* to extract the necessary values from a functor value. Our context-switch mechanism can be used to evaluate a functor, as shown in Fig. 21.

```
if OK and CurTerm.Kind = functorApp then
    if CurTerm.Argument.Value = undef then CurTerm := CurTerm.Argument
    elseif ReturnValue = undef then
        let functor = Lookup(CurEnv, CurTerm.Identifier).ValueEntry in
            EVALUATE(functor.FunctEnv,
                     CombineEnv(functor.FunctEnv,
                                CreateEnv(functor.FunctID,
                                          Restrict(CurTerm.Argument.Value,
                                                   functor.FunctSig))))
        endlet
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        CurTerm.Argument.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 21. Functor application.

5 ASM for SML: Matches

In this section, we focus on the evaluation of SML match rules. An SML match rule consists of a pattern and an associated expression. Match rules are always evaluated with respect to a target value. A successful attempt to match a value against a pattern results in an environment, representing the bindings required to match the value against that pattern; in such situations, the associated expression is then evaluated with respect to the current environment augmented by the new bindings, and its value returned. An unsuccessful attempt to match a value against a pattern results in the special value *Fail: Values*.

5.1 Match Lists

Matches usually occur in a list. Evaluating a list of matches is relatively straightforward; one evaluates each match in the list sequentially until a non-*Fail* value is generated, which is returned as the value of the list expression. Should all matches in the list return *Fail*, the list returns *Fail* as well.

The rule for evaluating match lists is shown in Fig. 22. Note that we associate with every match list its component rule (found with *MatchRule: Terms* → *Terms*) and the remainder of the list which follows that rule (found with *MatchList: Terms* → *Terms*).

---

if *OK* **and** *CurTerm.Kind = matchList* **then**
    if *CurTerm.MatchRule.Value = undef* **then** *CurTerm := CurTerm.MatchRule*
    elseif *CurTerm.MatchRule.Value* ≠ *Fail* **then**
        CurTerm.Value := CurTerm.MatchRule.Value
        CurTerm.MatchRule.Value := undef
        KEEPGOING
    elseif *CurTerm.MatchList = undef* **then**
        CurTerm.Value := Fail, CurTerm.MatchRule.Value := undef
        KEEPGOING
    elseif *CurTerm.MatchList.Value = undef* **then** *CurTerm := CurTerm.MatchList*
    else
        CurTerm.Value := CurTerm.MatchList.Value
        CurTerm.MatchList.Value := undef, CurTerm.MatchRule.Value := undef
        KEEPGOING
    endif
endif

Fig. 22. Match lists.

5.2 Match Rules

The semantics for evaluating match rules were explained in the introduction to this section; the corresponding rule is shown in Fig. 23.

5.3 Default Patterns

Here we begin to present rules for handling various types of patterns. Recall that *CurMatchVal* is used to hold the value against which the current pattern is to be matched; the result of a pattern match is either an environment or the special value *Fail*.

The simplest patterns to match are the underscore pattern "_" and the ellipsis pattern ". . .". Either pattern matches against any value and creates no bindings;

```
if OK and CurTerm.Kind = matchRule then
    if CurTerm.Pattern.Value = undef then CurTerm := CurTerm.Pattern
    elseif CurTerm.Pattern.Value = Fail then
        CurTerm.Value := Fail, CurTerm.Pattern.Value := undef
        KEEPGOING
    elseif ReturnValue = undef then
        EVALUATE(CurTerm.Expr, CombineEnv(CurEnv, CurTerm.Pattern.Value))
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        CurTerm.Pattern.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 23. Match rules.

the empty environment is returned. We use the distinguished element *EmptyEnv: Envs* to represent the environment containing no bindings. The corresponding simple rule is shown in Fig. 24.

```
if OK and (CurTerm.Kind = "_" or CurTerm.Kind = "...") then
    CurTerm.Value := EmptyEnv
    KEEPGOING
endif
```

Fig. 24. Default patterns.

5.4 Constant Patterns

One can match values against special patterns representing constants (*e.g.*, numeric literals). The match succeeds if and only if the constant pattern corresponds to the current match value; no bindings are generated. Using the function *ConstValue: Terms* $\rightarrow$ *Values* to indicate the true value of this constant term, the rule shown in Fig. 25 performs this operation.

5.5 Simple Identifiers

Matching a value against an identifier in the current environment succeeds in three cases:

1. The identifier is currently unbound. The resulting value is a single-entry environment, binding the identifier to the value.

if *OK* **and** *CurTerm.Kind* = *specialConstant* **then**
  if *CurMatchVal* = *CurTerm.ConstValue* **then** *CurTerm.Value* := *EmptyEnv*
  else *CurTerm.Value* := *Fail*
  endif
  KEEPGOING
endif

Fig. 25. Constant patterns.

2. The identifier is currently bound to a free variable. The resulting value is a single-entry environment, binding the identifier to the variable.
3. The identifier is already bound to the desired value. No additional bindings are generated.

We use a new function *CreateEnv: Identifiers* $\times$ *Values* $\rightarrow$ *Envs* to create a single-entry environment with the specified binding. The corresponding rule is shown in Fig. 26.

if *OK* **and** *CurTerm.Kind* = *identifier* **then**
  let *id* = *CurTerm.Identifier* **in**
    if *Lookup(CurEnv, id)* = *undef*
    or *Lookup(CurEnv, id).TagEntry* = *ValueVariable* **then**
      CurTerm.Value := *CreateEnv(id, Pair(CurMatchVal, ValueVariable))*
    elseif *Lookup(CurEnv, id).ValueEntry* = *CurMatchVal* **then**
      CurTerm.Value := *EmptyEnv*
    else *CurTerm.Value* := *Fail*
    endif
    KEEPGOING
  endlet
endif

Fig. 26. Identifier patterns.

5.6 Records: Labeled Patterns

Recall that the value of a record in SML is a finite name/value mapping. To match a record value against a sequence of labeled patterns, we simply need to ensure that the labeled sequence agrees with the record. That is, if label ℓ is associated with expression e, the current match value should associate ℓ with a value v such that v successfully (recursively) matches against e.

Consequently, our context-switch rules come in handy again, as we need to repeatedly check that each labeled expression successfully matches against the corresponding values in the match record. Our rule, shown in Fig. 27, simply proceeds along the list of labeled patterns, combining the resulting environments as needed.

---

if OK **and** $CurTerm.Kind = labeledPattern$ **then**
    if $ReturnValue = undef$ **then**
        $EVALUATE(CurTerm.Pattern,\ CurEnv,$
                $MapLookup(CurMatchVal,\ CurTerm.Label))$
    else
        if $ReturnValue = Fail$ **then**
            $CurTerm.Value := Fail$
            $KEEPGOING$
        elseif $CurTerm.MoreBindings = undef$ **then**
            $CurTerm.Value := ReturnValue$
            $KEEPGOING$
        elseif $CurTerm.MoreBindings.Value = undef$ **then**
            $CurTerm.Pattern.Value := ReturnValue$
            $CurTerm := CurTerm.MoreBindings$
        else
            if $CurTerm.MoreBindings.Value = Fail$ **then** $CurTerm.Value := Fail$
            else $CurTerm.Value := CombineEnv(CurTerm.Pattern.Value,$
                            $CurTerm.MoreBindings.Value)$
            endif
            $CurTerm.Pattern.Value := undef$
            $CurTerm.MoreBindings.Value := undef$
            $KEEPGOING$
        endif
    endif
endif

Fig. 27. Record matching.

5.7 Constructed Patterns

Constructed patterns are patterns consisting of a constructor and an argument. A constructed pattern successfully matches against a value which is itself a constructed value with the same identifier and whose value (recursively) matches against the pattern argument. The rule for constructed patterns is shown in Fig. 28.

5.8 Memory References

A memory reference term successfully matches against a value if the value is an address, and the value stored at that address in memory (recursively) matches

```
if OK and CurTerm.Kind = constructedPattern then
    let id = CurTerm.Identifier in
        if Lookup(id, CurEnv).ValueEntry.Second ≠ ValueConstructor
        and Lookup(id, CurEnv).ValueEntry.Second ≠ ExceptionConstructor then
            CurTerm.Value := Fail
            KEEPGOING
        elseif Lookup(id, CurEnv).ValueEntry.First ≠ CurMatchVal.First then
            CurTerm.Value := Fail
            KEEPGOING
        elseif ReturnValue = undef then
            EVALUATE(CurTerm.Argument, CurEnv, CurMatchVal.Second)
        else
            CurTerm.Value := ReturnValue, ReturnValue := undef
            KEEPGOING
        endif
    endlet
endif
```

Fig. 28. Constructed pattern matches.

against the argument of the reference term. The rule for memory reference terms is given in Fig. 29.

```
if OK and CurTerm.Kind = memoryReference then
    if ReturnValue = undef then
        EVALUATE(CurTerm.Argument, CurEnv, Store(CurMatchVal))
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        KEEPGOING
    endif
endif
```

Fig. 29. Reference matches.

5.9 Layered Patterns

Layered patterns are patterns of the form *id AS pattern*. The current match value is matched against the pattern argument; if successful, the match generates one additional binding: the specified identifier is bound to the current match value as a whole. The rule for layered patterns is given in Fig. 30.

if OK **and** $CurTerm.Kind = layeredPattern$ **then**
  if $CurTerm.Argument.Value = undef$ **then** $CurTerm := CurTerm.Argument$
  else
    if $CurTerm.Argument.Value = Fail$ **then** $CurTerm.Value := Fail$
    else
      $CurTerm.Value := CombineEnv(CurTerm.Argument.Value,$
        $CreateEnv(CurTerm.Identifier,\ Pair(CurMatchVal,\ ValueVariable)))$
    endif
    $CurTerm.Argument.Value := undef$
    $KEEPGOING$
  endif
endif

Fig. 30. Layered patterns.

6 ASM for SML: Declarations and Environments

In this section, we focus on the evaluation of SML declarations, which create and manipulate environments for use in evaluating other terms.

6.1 Global Declarations

SML contains several types of statements for creating value, datatype, and exception declarations. The lone argument in each case for such declarations is an expression which evaluates to an environment; the bindings in that environment are then added to the current environment and used in future evaluations. The ASM rule for handling declarations is shown in Fig. 31.

if OK **and** $CurTerm.Kind = declaration$ **then**
  if $CurTerm.Declaration.Value = undef$ **then** $CurTerm. := CurTerm.Declaration$
  else
    $CurEnv := CombineEnv(CurEnv,\ CurTerm.Declaration.Value)$
    $CurTerm.Declaration.Value := undef$
    $KEEPGOING$
  endif
endif

Fig. 31. Global declarations.

6.2 Datatype Equality Declarations

SML contains a statement for defining a datatype to be equal to another datatype already defined in the current environment. The semantics of this statement are simply illustrated in the ASM rule shown in Fig. 32; the new identifier is bound to the definition of the old identifier, and the new binding is added to the current environment.

---

if *OK* **and** *CurTerm.Kind = datatypeEquality* **then**
  CurEnv := CombineEnv(CurEnv,
    CreateEnv(CurTerm.Identifier,
      Lookup(CurEnv, CurTerm.Identifier)))
  KEEPGOING
endif

Fig. 32. Datatype equality declarations.

6.3 Local Declarations

Local declarations are a mechanism for defining environments while using declarations which are not part of the resulting environment, but used in the creation of that environment. Thus, local declarations contain two sets of declarations: a set of local declarations which will not be part of the final environment, and a set of declarations (usually using the local declarations) which become the resulting environment.

Thus, to evaluate a local declaration, we need to temporarily add the first set of declarations into the current environment while we evaluate the second set of declarations. Thus, we can again use our context-switch mechanism to perform this task. The corresponding ASM rule is shown in Fig. 33. (A similar mechanism is used for abstract types, sequential declarations, top-level declarations, and programs; the ASM rules are identical and we omit them.)

6.4 Open Statements

An open statement in SML takes a list of identifiers whose values in the current environment are themselves environments. The effect of the open statement is to add these environments to the current environment.

To represent open statements, we use a new universe of *Lists* (of values), a function *IdList: Terms* $\rightarrow$ *Lists* to indicate the list of identifiers associated with a given open statement, and the usual list functions *Nth: Lists* $\times$ *Naturals* $\rightarrow$ *Values* and *Length: Lists* $\rightarrow$ *Naturals*. The rule shown in Fig. 34 sequentially processes the list of identifiers.

6.5 Value Bindings

A value binding in SML takes the form "*pattern = expression*". To evaluate a value binding, one attempts to match the pattern against the value of the expression in the current environment (as with SML function calls). Recall that if the

```
if OK and CurTerm.Kind = localDeclaration then
    if CurTerm.LocalEnv.Value = undef then CurTerm := CurTerm.LocalEnv
    elseif ReturnValue = undef then
        EVALUATE(CurTerm.Declaration,
                    CombineEnv(CurEnv, CurTerm.LocalEnv.Value))
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        CurTerm.LocalEnv.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 33. Local declarations.

```
if OK and CurTerm.Kind = open then
    if Counter = 1 then
        CurTerm.Value := Lookup(CurEnv, Nth(CurTerm.IdList, Counter))
        Counter := Counter + 1
    elseif Counter ≤ CurTerm.IdList.Length then
        CurTerm.Value :=
            CombineEnv(CurTerm.Value,
                        Lookup(CurEnv, Nth(CurTerm.IdList, Counter)))
        Counter := Counter + 1
    else
        Counter := 1
        KEEPGOING
    endif
endif
```

Fig. 34. Open statements.

match succeeds, the result of evaluating a match is an environment; that environment becomes the value of the binding. If the match fails, a special exception (represented by *BindException: Exceptions*) is raised.

A value binding may be followed by more value bindings; the bindings are evaluated sequentially and combined appropriately. The rule for evaluating value bindings is shown in Fig. 35. (The rule for evaluating datatype bindings is essentially identical and we thus omit the rule.)

```
if OK and CurTerm.Kind = valueBinding then
    if CurTerm.Expr.Value = undef then CurTerm := CurTerm.Expr
    elseif CurTerm.Pattern.Value = undef then
        if ReturnValue = undef then
            EVALUATE(CurTerm.Pattern, CurEnv, CurTerm.Expr.Value)
        else
            if ReturnValue = Fail then
                CurTerm.Expr.Value := undef, CurException := BindException
            else CurTerm.Pattern.Value := ReturnValue
            endif
        ReturnValue := undef
        endif
    elseif CurTerm.MoreBindings = undef then
        CurTerm.Value := CurTerm.Pattern.Value
        CurTerm.Expr.Value := undef, CurTerm.Pattern.Value := undef
        KEEPGOING
    elseif CurTerm.MoreBindings.Value = undef then
        CurTerm := CurTerm.MoreBindings
    else
        CurTerm.Value := CombineEnv(CurTerm.Pattern.Value,
                                    CurTerm.MoreBindings.Value)
        CurTerm.Expr.Value := undef, CurTerm.Pattern.Value := undef
        CurTerm.MoreBindings.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 35. Value bindings.

6.6 Constructor Bindings

A constructor binding is simply an identifier which is to be considered as a value constructor for the duration of the program. Evaluating a constructor binding results in a one-entry environment indicating that the named identifier is a value constructor. As with other SML constructs, constructor bindings may occur in a list as well; the resulting environment contains all the named bindings. The ASM rule for evaluating constructor bindings is shown in Fig. 36.

---

if *OK* **and** *CurTerm.Kind = constructorBinding* **then**
  if *CurTerm.MoreBindings = undef* **then**
    CurTerm.Value := CreateEnv(CurTerm.Identifier,
        Pair(CurTerm.Identifier, ValueConstructor))
    KEEPGOING
  elseif *CurTerm.MoreBindings.Value = undef* **then**
    CurTerm := CurTerm.MoreBindings
  else
    CurTerm.Value :=
      CombineEnv(CreateEnv(CurTerm.Identifier,
          Pair(CurTerm.Identifier, ValueConstructor)),
        CurTerm.MoreBindings.Value)
    CurTerm.MoreBindings.Value := undef
    KEEPGOING
  endif
endif

Fig. 36. Constructor bindings.

6.7 Exception Bindings

Exception bindings are essentially the same as constructor bindings; the only difference is that the value of an exception name is guaranteed to be different from any other value used in the program (so that exceptions do not get confused with normal values). We use the ASM extend rule to generate a new value for the exception name to ensure this. The rule is shown in Fig. 37.

6.8 Functor Bindings

As mentioned above, a functor is a quadruple *(id, sig, exp, env)*. Evaluating a functor binding term results in the construction of a functor using the named identifier, signature, and expression term, using the current environment. As before, functor bindings may occur in a list of such bindings. We use a function *BuildFunctor: Identifiers* $\times$ *Signatures* $\times$ *Terms* $\times$ *Envs* $\rightarrow$ *Values* to construct the appropriate functor value. The ASM rule for functor bindings is shown in Fig. 38.

6.9 Programs and Exceptions

An SML program is simply a declaration, optionally followed by another program; thus, the rules given earlier for sequences of declarations usually suffice

```
if OK and CurTerm.Kind = exceptionBinding then
    if CurTerm.MoreBindings = undef then
        extend Exceptions with new
            CurTerm.Value := CreateEnv(CurTerm.Identifier,
                                       Pair(new, ExceptionConstructor))
        endextend
        KEEPGOING
    elseif CurTerm.MoreBindings.Value = undef then
        CurTerm := CurTerm.MoreBindings
    else
        extend Exceptions with new
            CurTerm.Value :=
                CombineEnv(CreateEnv(CurTerm.Identifier,
                                     Pair(new, ExceptionConstructor)),
                           CurTerm.MoreBindings.Value)
        endextend
        CurTerm.MoreBindings.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 37. Exception bindings.

```
if OK and CurTerm.Kind = functorBinding then
    if CurTerm.MoreBindings = undef then
        CurTerm.Value := CreateEnv(CurTerm.Identifier,
                                   BuildFunctor(CurTerm.FunctorID,
                                                CurTerm.Signature,
                                                CurTerm.Expr, CurEnv))
        KEEPGOING
    elseif CurTerm.MoreBindings.Value = undef then
        CurTerm := CurTerm.MoreBindings
    else
        CurTerm.Value :=CombineEnv(CurTerm.MoreBindings.Value,
                        CreateEnv(CurTerm.Identifier,
                                  BuildFunctor(CurTerm.FunctorID,
                                               CurTerm.Signature,
                                               CurTerm.Expr, CurEnv)))
        CurTerm.MoreBindings.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 38. Functor bindings.

for handling SML programs. The only difference deals with the treatment of exceptions. Exceptions do not propagate past the top-level declaration of a program; thus, if an exception ever reaches a program term, the normal exception processing is suspended and control passes to the next program in the sequence. This behavior is captured in the rule shown in Fig. 39.

if $CurException \neq undef$ **and** $CurTerm.Kind = program$ **then**
  $CurException := undef,\ CurTerm.LocalEnv.Value := EmptyEnv$
  KEEPGOING
endif

Fig. 39. Catching program-level exceptions.

7 Discussion

ASMs were first proposed as a methodology for specifying the semantics of programming languages 1984. ASMs have been applied to a wide variety of programming languages: imperative languages such as C/C++ 1993, 1995, logic programming languages such as Prolog (Börger and Rosenzweig 1994) and its variants, object-oriented languages such as Java (Börger and Schulte 1998, Wallace 1997) and Oberon (Kutter and Pierantonio 1997b), and hardware languages such as VHDL (Börger *et. al.* 1995). To the best of our knowledge, this case study in Standard ML is the first application of ASMs to provide the semantics of a functional programming language.

The official semantics of Standard ML is given in (Milner *et. al.* 1997), using an axiomatic semantics called Natural Semantics. The rules given in Milner, while definitive, rely heavily on axiomatic notation and proof rules which are difficult to read. With the presence of an official semantics, there appear to be few other treatments of SML using other semantic techniques. A recent paper (Watt 1999) announces the use of Action Semantics to give semantics to SML. Both of these works give static and dynamic semantics for SML.

One interesting feature of our ASM semantics for SML is our use of the EVALUATE macro, used to evaluate a term in an environment differing from the current environment. The EVALUATE macro appears in roughly one-third of the rules given above (not counting rules omitted for brevity), suggesting that term-environment evaluation plays an important role in the dynamic semantics of SML. This is natural; SML is, after all, a functional programming language, which draws its roots from the lambda calculus. Since function application (*i.e.*, term evaluation in a different environment) is at the heart of the lambda calculus, it seems natural that this feature should exhibit itself so prominently in our ASM semantics.

We would argue that the ASM dynamic semantics given above are as precise and accurate as conventional semantic treatments such as the official definition (Milner *et. al.* 1997). While the rules and explanations above are somewhat longer, the readability of these rules is substantially better. Such explanations can provide a better basis for explaining and understanding the language with a minimal amount of notational overhead.

Several possible extensions of this work are being contemplated. One clear gap in the descriptions above is any treatment of the static semantics of SML, including the extensive type system. The official definition of SML (Milner *et. al.* 1997) spends extensive time defining the static aspects of the language, which can also be confusing at times. Recently Montages (Kutter and Pierantonio 1997a) have been used to describe both static and dynamic semantics of programming languages using ASMs; we consider extending this work using Montages to provide static semantics for SML as well. Alternatively, ASMs have been used for various proofs of correctness, such as compilation techniques; we consider specifying a compilation technique for SML to some intermediate language and proving its correctness.

References

Börger, E., U. Glässer, and W. Muller, "Formal Definition of an Abstract VHDL'93 Simulator By EA-Machines." In C. Delgado Kloos and P. T. Breuer, eds., *Formal Semantics for VHDL*, 107–139. Kluwer Academic Publishers, 1995.

Börger, E., and J. Huggins, "Abstract State Machines 1988-1998: Commented ASM Bibliography", Formal Specification Column (H. Ehrig, ed.), EATCS Bulletin 64, February 1998, 105–127. (An updated version is available from (Huggins 1999).)

Börger, E., and D. Rosenzweig, "A mathematical definition of full Prolog." In *Science of Computer Programming*, vol. 24, 249–286. North-Holland, 1994.

Börger, E., and W. Schulte, "A Programmer Friendly Modular Definition of the Semantics of Java." In J. Alves-Foss, ed., *Formal Syntax and Semantics of Java*, Springer Lecture Notes in Computer Science. Springer-Verlag, 1998.

Glässer, U., editor, Abstract State Machines Europe Home Page, `http://www.uni-paderborn.de/cs/asm/`

Gurevich, Y., "Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs." Technical Report CRL-TR-38-84, EECS Department, University of Michigan, 1984.

Gurevich, Y., "Logic and the Challenge of Computer Science." In E. Börger, editor, *Current Trends in Theoretical Computer Science*, 1–57. Computer Science Press, 1988.

Gurevich, Y., "Evolving Algebras: An Attempt to Discover Semantics". In G. Rozenberg and A. Salomaa, eds., *Current Trends in Theoretical Computer Science*, 266–292. World Scientific, 1993. (First published in *Bulletin of EATCS 43* (Feb.), pages 264-284.)

Gurevich, Y., "Evolving Algebras 1993: Lipari Guide." In E. Börger, editor, *Specification and Validation Methods*, 9–36. Oxford University Press, 1995.

Gurevich, Y., "The Sequential ASM Thesis." Microsoft Research Technical Report MSR-TR-99-09, 1999.

Gurevich, Y., and J. Huggins, “The Semantics of the C Programming Language.” In *Selected papers from CSL’92 (Computer Science Logic)*, Springer Lecture Notes in Computer Science 702, 274–308. Springer-Verlag, 1993.

Huggins, J., editor, Abstract State Machines Home Page, `http://www.eecs.umich.edu/gasm/`

Kutter, P., and A. Pierantonio, “Montages: Specifications of Realistic Programming Languages”, *Journal of Universal Computer Science*, vol. 3, no. 5 (1997), 416–442.

Kutter, P., and A. Pierantonio, “The Formal Specification of Oberon”, *Journal of Universal Computer Science*, vol. 3, no. 5 (1997), 443–503.

Lucent Technology, Standard ML of New Jersey Home Page, `http://cm.bell-labs.com/cm/cs/what/smlnj/`

Milner, R., M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

Wallace, C., “The Semantics of the C++ Programming Language.” In E. Börger, editor, *Specification and Validation Methods*, 131–164. Oxford University Press, 1995.

Wallace, C., “The Semantics of the Java Programming Language: Preliminary Version.” Technical Report CSE-TR-355-97. EECS Department, University of Michigan, 1997.

Watt, D., “The Static and Dynamic Semantics of Standard ML”, in Proceedings of the Second International Workshop on Action Semantics (AS’99), BRICS Notes Series NS-99-3, Department of Computer Science, University of Aarhus, May 1999, 155–172.