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. The Abstract State Machines (ASM) methodology is 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 (appropriately pre-processed) Standard ML programs; the effect of a Standard ML instruction can be seen in terms of the corresponding actions performed by the ASM.

1 Introduction

The Abstract State Machines (ASM) methodology [Gur95] is a methodology for formally specifying computing systems (software, hardware, or mixed). First introduced by Gurevich [Gur88] (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 [BH98,Glä,Hug]. The ASM Thesis [Gur00] 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." [Luc]

In this paper, we describe the dynamic semantics of SML using the ASM methodology, using Milner [MTHM97] 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 [MTHM97] 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. (For example, arbitrary tuples of the form $(t_1, t_2, \ldots, t_n)$ are translated into records

of the form $\{1 = t_1, 2 = t_2, \ldots, n = t_n\}$.) 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.

For brevity, we omit discussion of SML constructs dealing with structure restrictions, functor applications, layered patterns, declarations, and value bindings; a full discussion of these constructs may be found in [CH99].

We assume familiarity with sequential ASMs in the following sections; see [Gur91] for an introduction to sequential ASMs.

2 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.

2.1 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.)

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

Fig. 1. Evaluating constants.

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* → *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 in our rules. *OK* abbreviates the term "*CurException = undef*"; it indicates that the computation in process is proceeding normally (*i.e.*, no exception is present). *KEEPGOING* abbreviates the rule "*CurTerm := CurTerm.Next*"; it is 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.

2.2 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* → *Values* to map a constant expression to its corresponding value. The corresponding ASM rule, shown in Fig. 1, is straightforward.

2.3 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

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

Fig. 2. Identifiers.

of *Identifiers* to represent these names. The binary function *Lookup: Envs* × *Identifiers* → *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* → *Values* and *TagEntry: EnvEntries* → *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* → *Identifiers*) and extracts the corresponding value from the current environment. The rule is shown in Fig. 2.

2.4 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* × *Values* → *Maps* and *AddToMap: Maps* × *Identifiers* × *Values* → *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. 3) move through this sequence of record expressions, constructing the corresponding map. (The rule below uses functions *Expr: Terms* → *Terms* and *NextRecord: Terms* → *Terms* to indicate the two components of a record expression; we implicitly introduce component functions such as these in future rules without comment.)

2.5 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 en-

```
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. 3. Records.

```
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. 4. Reference terms.

vironment 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* → *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. 4.

```
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. 5. Assignment expressions.

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

2.7 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*).

Exceptions propagate by moving from child term to parent term repeatedly until a `handle` expression is found. The rules shown in Fig. 6 show how exceptions are raised and propagated. Note that the presence of a propagating exception in *CurException* falsifies the *OK* term, thus making most other rules inapplicable in this situation.

3 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.

```
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
if CurException ≠ undef and CurTerm.Kind ≠ handle then
    CurTerm.Value := undef, CurTerm := CurTerm.Parent
endif
```

Fig. 6. Raising and propagating exceptions.

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

Fig. 7. Function closures.

3.1 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* × *Envs* → *FunctionClosures* and projection functions *MatchBinding: FunctionClosures* → *Terms* and *EnvBinding: FunctionClosures* → *Envs*. The rule for evaluating function closure expressions is shown in Fig. 7.

3.2 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. 8 performs this evaluation in all cases.

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

```
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. 8. Evaluating arguments for function application.

```
if OK and CurTerm.Kind = functionApp and CurTerm.RightExpr.Value ≠ 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. 9. Constructor application.

use the function *Pair: Values* × *Values* → *Values* to construct such pairs. The rule for constructor application is shown in Fig. 9.

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* × *Values* → *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. 10 checks the resulting value before passing it along.

3.3 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

```
if OK and CurTerm.Kind = functionApp and CurTerm.RightExpr.Value ≠ 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. 10. Primitive operator application.

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 [GH93]), we use the universe *Naturals* of natural numbers and redefine the functions named above to have the following signatures:

Value: Terms × *Naturals* → *Values* *CurTerm: Naturals* → *Terms*
CurException: Naturals → *Exceptions* *CurEnv: Naturals* → *Envs*

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. In order to simplify our presentation (and following the precedent in [BR94]), 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).

```
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. 11. *EVALUATE* abbreviation.

```
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. 12. Function application to function closures.

We introduce some functions used in the process of making a context-switch dealing with the call stack. *Target: Naturals* $\rightarrow$ *Terms* indicates the top-level term which is to be evaluated at this level of the stack. *CurMatchValue: Naturals* $\rightarrow$ *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. 11. 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.

Finally, we can present the rule for performing a function application to a function closure. The rule shown in Fig. 12 simply invokes the specified function closure, attempting to match the specified argument against the function closure term.

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

Fig. 13. New definition of the *KEEPGOING* abbreviation.

```
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. 14. Returning from a function call.

3.4 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. 13; it transfers control in such situations to a new distinguised element *ReturnTerm: Terms*, where we perform the actual return.

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. 14.

3.5 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. 15; notice that the truth of the guard

```
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. 15. Normal processing of handle expressions.

```
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. 16. Exception processing of handle expressions.

OK indicates that no exceptions have occurred while processing the argument term.

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. 16.

3.6 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.

```
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. 17. Let expressions.

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. 17.

4 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*.

4.1 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. 18. Note that we associate with every match list its component rule (found with *MatchRule: Terms* $\rightarrow$ *Terms*) and the remainder of the list which follows that rule (found with *MatchList: Terms* $\rightarrow$ *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. 18. Match lists.

4.2 Match Rules

The semantics for evaluating match rules were explained in the introduction to this section; the corresponding rule is shown in Fig. 19.

4.3 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; 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. 20.

4.4 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. 21 performs this operation.

```
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. 19. Match rules.

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

Fig. 20. Default patterns.

4.5 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.
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.

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

Fig. 21. Constant patterns.

```
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. 22. Identifier patterns.

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. 22.

4.6 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. 23, simply proceeds along the list of labeled patterns, combining the resulting environments as needed.

4.7 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. 24.

4.8 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 against the argument of the reference term. The rule for memory reference terms is given in Fig. 25.

```
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. 23. Record matching.

5 Discussion

ASMs were first proposed as a methodology for specifying the semantics of programming languages [Gur84]. ASMs have been applied to a wide variety of programming languages: imperative languages such as C/C++ [GH93,Wal95], logic programming languages such as Prolog [BR94] and its variants, object-oriented languages such as Java [BS98,Wal97] and Oberon [KP97b], and hardware languages such as VHDL [BGM94]. 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 by Milner [MTHM97], using an axiomatic semantics called Natural Semantics. The rules given in Milner, while definitive, rely heavily on axiomatic notation and proof rules which can be 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 by Watt [Wat99] announces the use of Action Semantics to give semantics to SML; another work by Harper and Stone [HS98] translates SML into a typed

```
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. 24. Constructed pattern matches.

```
    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. 25. Reference matches.

variant of the lambda calculus to which an operational semantics is given. All of these other works give both 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 [MTHM97]. The rules and explanations above are somewhat longer than in an axiomatic approach, as the ASM notation lends itself to natural language notations rather than axiomatic proof rules. However, one may find 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 [MTHM97] spends extensive time defining the static aspects of the language, which can also be confusing at times. Recently Montages [KP97a] 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.

Extending this work to the Montages framework has another important benefit: executability. At this time, the ASM description given here has not been tested by any executable tools. The Gem-Mex tool [AKP97] allows one to execute Montage descriptions directly; we expect that doing so will give us further evidence of the correctness of these descriptions.

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

[AKP97] M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.

[BGM94] E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE CS Press.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998. (An updated version is available from [Hug].).

[BR94] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[BS98] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[CH99] S. Cater and J. Huggins. An ASM Dynamic Semantics for Standard ML. Technical Report CPSC-1999-2, Kettering University, 1999.

[GH93] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[Glä] U. Glässer. Abstract State Machines Europe Home Page. http://www.uni-paderborn.de/cs/asm/.

[Gur84] Y. Gurevich. Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs. Technical Report CRL-TR-38-84, EECS Department, University of Michigan, 1984.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[Gur91] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991. (Reprinted in G. Rozenberg and A. Salomaa, eds., *Current Trends in Theoretical Computer Science*, World Scientific, 1993, 266–292.).

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur00] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, page to appear, 2000.

[HS98] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998.

[Hug] J. Huggins. Abstract State Machines Home Page. http://www.eecs.umich.edu/gasm/.

[KP97a] P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[KP97b] P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

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

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Wal95] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[Wal97] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.

[Wat99] D. Watt. The Static and Dynamic Semantics of Standard ML. In *Proceedings of the Second International Workshop on Action Semantics (AS'99)*, number NS-99-3 in BRICS Notes Series, pages 155–172. Department of Computer Science, University of Aarhus, May 1999.