Reasoning with the Formal Definition of Standard ML in HOL

Donald Syme
Department of Computer Science
The Australian National University
G.P.O. Box 4, Canberra, ACT, 2601, Australia
E-Mail: donald@cs.anu.edu.au

May 1, 1993

Abstract

This paper describes the results of a project to embed the Formal Definition of the Standard ML Core language into the HOL mechanized reasoning system. HOL theories of the Core syntax and dynamic semantics are presented, including a purely definitional representation of the semantic inference rules. The correctness of the representation is demonstrated by the derivation of several important language properties, including determinacy. A symbolic evaluator to assist in proving properties of Standard ML program fragments is also described. Some sample applications of the theory in the area of program verification are given. A recurring theme in the work is the need for logical precision in all aspects of a HOL representation.

1 Introduction

The Formal Definition of Standard ML was completed by Milner, Tofte and Harper in 1990 [14]. The aim of the definition was to give a rigorous and complete mathematical description of all aspects of the language, from its syntax to the semantics of interactive operation. The Definition describes the language on three levels - the Core language, the Module system and Programs. In turn each of these sections is described on three levels: *syntax*, *static semantics* and *dynamic semantics*. The main concern of this paper is with the dynamic semantics of the Core language, that is to say, with the runtime semantics of the basic constructs of any Standard ML program. Little substantial reasoning can be performed on the dynamic aspects of a Standard ML program without an adequate representation of these semantics.

	Core	*Modules*	*Programs*
Syntax	×	×	×
Statics	×	×	×
Dynamics	•	×	×

The semantics of the Core language are described in the Formal Definition using Natural Semantics [14, 11, 9]. The basic constructs in the semantics are sentences of the form $A \vdash phrase \Rightarrow R$. A sentence $A \vdash phrase \Rightarrow R$ denotes that in some background A the execution of *phrase* will give the result R. A set of *inference rules* define which sentences are inferable. The inference rules are the heart of the semantics, and take the form

$$\frac{A_1 \vdash phrase_1 \Rightarrow R'_1 \ \cdots \ A_n \vdash phrase_n \Rightarrow R'_n}{A \vdash phrase \Rightarrow R'}$$

There are 57 inference rules in the Core dynamic semantics, each of which deals with a particular case of evaluation.

In the case of the Core language, the background and result objects (A and R) are composed of standard semantic constructs such as environments E, states s, values *val*, closures C, exception packets p and records *rec*. Phrases are made up of syntactic fragments of Standard ML code.

The other major aspect of the dynamic semantics to be modeled in this work is the semantics of the fundamental values provided by Standard ML. These include arithmetic and comparison operators such as `+`, `-` and `=`. The semantics of the standard I/O facilities and of real numbers are omitted.

Section 2 of this paper deals with the construction of HOL representations for the syntactic and semantic types used in the Definition. Section 3 deals with the construction of a representation for the inference rules and other semantic notions. In Section 4 a series of useful properties are proven in HOL based on the theories developed in Sections 2 and 3. Section 5 describes the construction of a symbolic evaluator for the Core language. Section 6 describes a mechanism for reasoning cleanly about concretely-typed Standard ML values within HOL based theorem proving. Lastly, in Section 7 the theory and tools constructed are applied to several areas. Correctness theorems for two pieces of Standard ML code are stated and verified in HOL, some computation properties such as termination and no-state-change are defined and used, and the equivalence of two small program fragments is demonstrated.

This paper is a summary of subthesis completed by Donald Syme in partial fulfillment of the requirements of an Honours degree at the Australian National University in 1992. Some details have been omitted, and a full description of the work can be found in [16]. That subthesis was based in part on work performed by Matthew Hutchins for his Honours project in 1990, also at the Australian National University [7].

2 Embedding the Types in HOL

The first task to be faced in embedding the semantics of the Core into HOL is to construct a representation of the syntactic and semantic types that are used in the Core semantics.

2.1 Representing the Syntactic Types

A small subset of the Standard ML syntax is shown below.

AtExp	=	AtExp_Const *Const*
	|	AtExp_Paren *Exp*
Exp	=	Exp_AtExp *AtExp*
	|	Exp_Const *Const*

Constructing the HOL representations of these types is conceptually quite easy, as concrete syntax trees are well understood and to some extent supported in HOL. The Standard ML syntax, however, poses one substantial problem - it is a mutually recursive syntax, and no automated tools are built into HOL for the definition of recursive grammars. A package written in HOL90 by Claudio Russo of Edinburgh [15] was ported to HOL88 and used to define both the types themselves and various mutually recursive functions over the types. Making the definitions introduces 14 new HOL types, including *Exp*, *At_Exp*, *Pat*, *Const* and *Phrase*.

2.2 Representing the Semantic Types

The semantic types pose more substantial problems for the type definition mechanisms of the HOL system. Not only are the semantic types mutually recursive, but they are also non-concrete, as they contain *finite maps*. In the Core semantics, finite maps are used to represent records, memory and environments. Finite maps are not directly supported in HOL, and so a *finite_map* theory was created. This theory is in the spirit of the *finite_set* theory already present in the HOL system. Finite maps are also known as finite partial functions or lookup tables. The theory constructed for finite maps contains operators such as DOMAIN, RANGE, APPLY, EXT (for extension of the map) and TRANSFORM. A substantial number of useful theorems were proven about these operators, including useful results allowing induction over the domain of the map. For example, the following theorems demonstrate basic results about applying the map:

$$\begin{array}{l}
\vdash \texttt{APPLY}\ z\ \texttt{ZIP} = \texttt{FAILURE} \\
\vdash \texttt{APPLY}\ z\ (\texttt{EXT}\ (z,v)\ f) = \texttt{RESULT}\ v \\
\vdash \texttt{APPLY}\ z\ (\texttt{EXT}\ (x,v)\ f) = \\
\quad ((x = z) \rightarrow \texttt{RESULT}\ v \mid \texttt{APPLY}\ x\ f)
\end{array}$$

Finite maps are essentially a HOL construct used to avoid the syntactic and semantic obfuscation involved with using functions or lists to represent these objects. They do not solve the fundamental problem involved with the mutual recursion present within the non-concrete semantic types. A proper solution to this latter problem involves type theory beyond the scope of this project and in the end a reasonable axiomatization of the recursive properties of the types was used. This axiomatization was based on small examples derived by other HOL users [6]. Significant steps were made toward justifying this axiomatization via a fully definitional approach.

For both the syntactic and semantic types a full pretty printer and parser were constructed, both of which were invaluable in later work. The pretty-printer was implemented by modifying the appropriate HOL88 Lisp code to gain maximum output efficiency.

The following illustrates the use of some semantic type constructors to make a *Val* which is a *Closure* consisting of a match and two environments.

```
# let tm = "Val_Closure (Closure m e v)";;
tm = "[m,e,v]" : term
    %< pretty-printed version of a closure >%
```

```
# let tm2 = dest_Val_Closure tm;;
tm2 = "Closure m e v" : term

# let tm3 = dest_Closure tm2;;
tm3 = ("m", "e", "v") : (term # term # term)
```

3 Embedding the Semantics in HOL

3.1 Incorrect Approaches

The second stage in embedding the Core language definition into HOL involves constructing appropriate representations of the semantic inference rules and some associated constructs.

A first attempt at creating a theory capturing these constructs was made by Matthew Hutchins at the Australian National University in 1990 [7]. He used an axiomatic approach to model the logical properties of the inference rules of the semantics. Initially each inference rule was asserted as an implicative axiom. Thus a simple rule, such as that for parenthesized expressions:

$$\frac{s, E \vdash exp \Rightarrow v, s'}{s, E \vdash (exp) \Rightarrow v, s'}$$

was asserted as the axiom

$$\vdash \mathtt{INFERABLE}(s,\ E,\ atexp,\ val,\ s') \supset$$
$$\mathtt{INFERABLE}(s,\ E,\ \texttt{(}\ atexp\ \texttt{)},\ val,\ s')$$

Here INFERABLE stands for the inferability of a particular sentence. A similar approach has been used in other verification environments based on the Formal Definition of Standard ML [3]. However, it soon becomes obvious within such schemes that important properties cannot be proven. The axioms do not fully capture the fact that a sentence is inferable *if and only if* it can be inferred via one of the inference rules. Within such an axiomatization scheme there is no way to prove that a sentence is *not* inferable, and thus there is no guarantee of determinacy or any other such reasonable language property.

After these problems were identified, Hutchins used another set of axioms which he thought corrected this problem. However, in this new scheme the same essential problem became manifest with infinite computations - there was no way of proving whether a sentence involving an infinite computation was or was not inferable. This effectively equates to the lack of a principal of computational induction. The incompleteness of the axiomatization was by no means easy to spot, and points to the inherent dangers in using axiom based approaches.

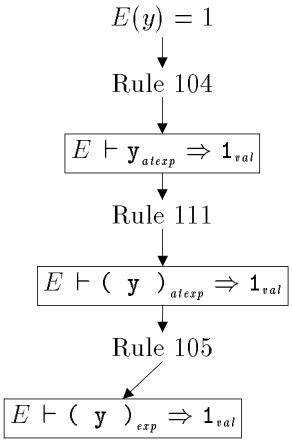

Figure 1: Inference Tree for "(y)"

3.2 A Correct Representation

Full tools are now becoming widely used for inductively defining relationships in HOL, and it has been demonstrated that it is possible to use this method to define relationships derived from a Natural Semantics definition [8, 12]. However, in this project a different, definitional approach was used to construct a correct representation of the semantics, based on the notion of *inference trees*. As noted in the Commentary on the Formal Definition of Standard ML, successive applications of the inference rules may be thought of as constructing an *inference tree* for a sentence [11]. The children of each node in an inference tree are the sub-evaluations that contribute to the evaluation of the phrase at the node. Thus an inference tree is a tree labeled with sentences where each sentence is directly deducible via inference rules from its children. A small inference tree is shown in Figure 1 for the expression "`( y )`" where `y` has value 1.

Inference trees can be used to define inferability in the following way:

Definition 1 *A sentence is inferable if and only if there exists a valid finite inference tree whose root node is that sentence.*

Validity of an inference tree is defined as follows:

Definition 2 *Given an inference tree T with root node*

$$s, obj \vdash phrase \Rightarrow res, s'$$

and children $T_1, \ldots, T_n$, then T is valid if and only if its node can be deduced via an inference rule from its children, and each if its children are also valid.

Thus, an inference tree is valid if and only if it represents a valid computation according to the inference rules.

Inference trees are modeled in HOL using the *ltree* theory and a set of definitions which capture the logical content of each individual inference rule. As an example, the predicate which tests for validity at an node containing a single variable is:

$$\vdash_{def}\ \texttt{VALID\_INFR\_Var}\ children\ (s, E, var, val, s') = \\ (children = \texttt{[]}) \wedge \\ (s' = s) \wedge \\ (E(var) = \texttt{RESULT}\ val)$$

A similar predicate is defined for each type of phrase in the syntax, and these are collected into the predicate VALID_NODE. The definitions of validity and inferability can then be made:

$$\vdash_{def}\ \texttt{VALID\_TREE (Node sent}\ ch) = \\ \texttt{VALID\_NODE (MAP ROOT}\ ch)\ \texttt{sent} \wedge \\ \texttt{EVERY VALID\_TREE}\ ch$$

$$\vdash_{def}\ \texttt{INFERABLE}\ sent = \\ \exists tree.\ \texttt{VALID\_TREE}\ tree \wedge (\texttt{ROOT}\ tree = sent)$$

There is an obvious one-to-one correspondence between sequences of inferences based on the inference rules and inference trees. Thus the definition of inferability given above precisely captures the entire set of inferable sentences specified by the definition. All the reasoning about the Core language in the following sections is based on the definition of inferability given above. In those sections a large number of more convenient results about inferability will be derived. Ultimately these eliminate the need to refer directly to inference trees at all.

3.3 Other Aspects of the Representation

Eliminating Free Choice

The dynamic semantics of the Core given in the Formal Definition have the unfortunate property that the are inherently non-deterministic. This is due to the fact that at certain times a free choice is allowed of names in a name-space. The semantics make no demand on the particular name chosen, except that it may not currently be in use. To model this notion, it was decided to ensure determinism and yet still capture the idea that no specific choice is enforced. In HOL this is done by using the free yet deterministic Hilbert-choice operator.

$$\vdash \texttt{CHOOSE\_Name}\ used\_names = \varepsilon n.\ \neg(n\ \texttt{IN}\ used\_names)$$

The essential property is that no knowledge is available of the exact choice made, and this is still maintained.

The Dynamic Basis

The basic Standard ML arithmetic and comparison operators are defined in the HOL theory by reference to the corresponding HOL operators and types. For example, the semantics of the + operator, which operates on a paired value, are defined by:

```
⊢def plus val =
     let n = MLNUM_TO_NUM (MLPAIR_FST val)
     and m = MLNUM_TO_NUM (MLPAIR_SND val) in
       OUT_OF_RANGE(n+m) →
          Packet `Sum`
        | NUM_TO_MLNUM (n + m)
```

With the definitions of the basic Standard ML operations in place, the definition of the APPLY function specified in Appendix B of the Formal Definition can be made:

```
⊢def APPLY BasVal_plus val = plus val ∧
     APPLY BasVal_minus val = minus val ∧
                    ...
     APPLY BasVal_geq val = geq val
```

4 Proving Language Properties

In one sense a complete HOL theory of Standard ML has now been constructed, for *any* fact which may be logically deduced from the Core dynamic semantics can now be proven in HOL. The semantics are essentially captured in terms of the INFERABLE predicate.

It is useful at this time to draw a parallel with reasoning about the natural numbers. The HOL type *num* is defined in terms of 0 and SUC. However, it is essential that this primitive theory be enriched with theorems and a whole range of derived results, some of which will be interesting in themselves and others of which are merely building blocks for other results. This is effectively the aim of the following section - to expand the present theory of Standard ML with theorems, rules, tactics and definitions.

4.1 Backward Rewrite Rules

The first set of results to be proven are those which describe exactly under what conditions a sentence may be inferred, in terms of its sub-inferences. These results are crucial in eliminating the need to constantly refer back to inference trees in our reasoning.

The following HOL goal captures the notion that the sentence $s, E \vdash var \Rightarrow val, s'$, which represents the evaluation of a variable, may be inferred if and only if the variable itself evaluates to the value *val* in the environment *E*, and if *s = s'*:

```
"INFERABLE(s, E, AtExp_Var var, val, s')  =
     (s' = s) ∧
     (E(var) = RESULT val)"
```

This result is proved quite easily by expanding the definition of INFERABLE. Note what the result is saying - that a sentence with a variable as its phrase can be inferred if and only if the conditions on the right hold.

There are 43 such rules, one for each type of phrase in the Standard ML syntax, with most of them being far more complex than that above. The proofs of all of these are streamlined through one tactic PROVE_BWDRULE_TAC. Once the results are proven they form the basis for a range of tactics and rules such as BWDRULE_REWRITE_TAC, which automatically determines the correct rule to apply in a given situation.

4.2 Forward Inference Rules

The natural way to think about an inference rule is as an implicative theorem, where the premises of the rule imply the conclusion. This is how the inference rules were originally modeled as axioms in the previous section. There are 86 forward inference rules in total, corresponding to the inference rules in the Formal Definition. For example, one of the rules for parenthesized expressions is:

```
⊢ INFERABLE(s, E, atexp, val, s') ⊃
  INFERABLE(s, E, ( atexp ), val, s')
```

The proofs of these results follow fairly easily from the backward rewrite rules. The forward inference rules are crucial in allowing the inferability of sentences to be proven, and they form the basis of the symbolic evaluator described in the following section.

4.3 The Pattern Matching Theorem

We now proceed to outline two properties of the Standard ML language, both of which are described in the Commentary on the Formal Definition [11]. These are the Pattern Matching theorem and the Determinacy theorem. Both of these properties have been proven in HOL using the theories constructed in Section 2 and 3.

As with many functional languages, pattern matching lies at the heart of function application in Standard ML. Pattern matching should have no side-effects, as no expressions are evaluated. The Commentary on Standard ML states the property in the following way:

Theorem 1 (Pattern Matching) *Let E, v and pat be any environment, value and pattern. Suppose that*

$$s, E, v \vdash pat \Rightarrow r, s'$$

can be inferred for states s, s' and result r. Then $s = s'$.

That is to say, no state change occurs due to the evaluation. Stating this as a HOL goal:

```
"INFERABLE(s, E, val, pat, res, s') ⊃  (s = s')"
```

This goal was proven by structural induction over the syntax of patterns. This induction employs the mutual recursion theorems from the Standard ML syntax. Each step of the induction holds because of the nature of the kind of inferences that can be made on that particular type of pattern.

4.4 The Determinacy Theorem

The second of the two theorems is perhaps the single most useful result proved in this project. It is the Determinacy Theorem, which states that any two computations over the same phrase starting in the same background will produce the same result. After due consideration, it becomes obvious from the Formal Definition that the evaluation process is in fact deterministic, apart from the free choice mentioned in Section 3. The theorem can be stated as follows:

Theorem 2 (Determinacy) *Let the two sentences*

$$s, A \vdash phrase \Rightarrow R, s'$$

$$s, A \vdash phrase \Rightarrow R', s''$$

both be inferred. Then $R = R'$ and $s' = s''$.

The proof of this theorem uses computational induction on the length of the inference process. The goal for the Determinacy Theorem is quite simple to state in HOL:

```
"INFERABLE(s, env, ph, res, s') ∧
 INFERABLE(s, env, ph, res', s'') ⊃
    (res = res') ∧ (s' = s'')"
```

The goal asserts the equality of the results and final states of the two inferable sentences.

Any proof of this theorem must use some form of *computational* induction. In the theory the finiteness of computation is captured by the finite size of the inference trees on which the definition of inferability is based. Computational induction is thus equivalent to an induction on the depth of the inference trees involved.

The proof of the theorem in HOL is very lengthy and can only be performed by automating each of the many sub-cases of the induction. It is an excellent example of a theorem that can only be formally proven by machine.

The Significance of Determinacy

The Determinacy Theorem ensures that each evaluation has *at most* one result. This result may not be strengthened to *exactly* one result, for two reasons. Firstly, there are infinite computations for which no result can be inferred. Secondly, many computations do not make sense under the typing rules of the language, and these in general have no valid inferences.

Determinism thus justifies the following definition of evaluation as a partial function:

```
⊢def  EVAL (s,E) phrase =
    (∃res s'. INFERABLE(s,E,phrase,res,s') →
       RESULT ε res,s'. INFERABLE(s,E,phrase,res,s') |
       FAILURE
```

The following theorem, which captures the intuitive relationship between evaluation and inferability, can then be proven:

```
⊢ INFERABLE(s,E,phrase,res,s') =
   (EVAL (s,E) phrase = RESULT (res,s'))
```

5 The Symbolic Evaluator

5.1 Basic Symbolic Evaluation

Evaluation of languages is, of course, a well understood process. As well demonstrated by Boyer and Moore [2], evaluation is a natural technique for reasoning about the properties of functional programs. The usefulness of evaluation is increased when it is conducted *symbolically*. In this way an evaluator can be used to prove properties of infinite families of computations. As an example, the expression `"let val x = ` n ` in 1 end"` will always evaluates to the value 1 regardless of n. By symbolic evaluation the inferability of the sentence $s, E \vdash$ `"let val x = ` n ` in 1 end"` $\Rightarrow 1, s$ can be proven.

A fully symbolic evaluator based on the Core semantics developed in the previous sections was implemented in HOL as part of this project. As the evaluator proceeds through the execution tree, theorems about the evaluation are proven via the forward inference rules derived in the previous section. A example of the invocation of the symbolic evaluator to prove the inferability of the sentence shown above is demonstrated below. Here P program fragment `"let val x = ` n ` in 1 end"`.

```
# SML_EVAL (s,E) P;

  ⊢ INFERABLE(s, E, P, 1, s)
  ⊢ EVAL(s, E) P = 1, s
#
```

5.2 Multiple Evaluation Paths

Often the symbolic evaluator will come to an evaluation which it cannot compute. One technique for handling this is to split the computation into two or more paths, each of which makes a different assumption about the symbols involved. For instance, when evaluating the match M = `true => false | false => true`, against an arbitrary value v, the evaluator produces three different theorems:

```
(v = true) ⊢
      EVAL (s, E, v) M = false, s
¬(v = true) ∧ (v = false) ⊢
      EVAL (s, E, v) M = true, s
¬(v = true) ∧ ¬(v = false) ⊢
      EVAL (s, E, v) M = [Match], s
```

5.3 Other Features of the Evaluator

The symbolic evaluator is capable of using extra information to aid it in the evaluation process. In particular, it accepts a list of theorems of the form

$$\vdash \forall v1..vn.\ \texttt{EVAL}\ (s,E)\ phrase = res, s'.$$

which provide the evaluator with particular evaluation results. Before the evaluator attempts the evaluation of any phrase it tries to instantiate one of these theorems to give an answer for the evaluation being considered. One main use of this facility in section 7 is to provide induction assumptions to the evaluator.

The evaluator is designed to allow extensions to be made whenever new and more complex evaluation patterns need to be added. For instance, a module was added to allow variables to be evaluated in partially undefined environments . This module was based on results proven about *finite_maps*. Another module was added to allow the evaluation of phrases involving the Core base values such as `+` and `=`.

The evaluator is also capable of reasoning about exception-producing expressions, and if the result of a computation is unknown it will split into two paths according to whether it assumes an exception has occurred or not.

5.4 An Evaluation Tactic

It is very useful to define an HOL tactic `EVAL_TAC` which invokes the evaluator in an attempt to solve a particular goal. The goal is expected to be of the form

$$\texttt{"EVAL}\ (s,\ E)\ phrase = res,\ s'\texttt{"}$$

`EVAL_TAC` supplies the arguments s, E and *phrase* to the evaluator, and if enough information is available the evaluator will return the evaluation theorem which can then be used to help prove the goal.

6 Reasoning with Standard ML Types

It is now almost possible to begin applying the theories and tools developed in the previous chapter to more substantial problems. However, any attempts to do so run into immediate difficulties due to an inability to talk about typed values in any systematic manner. In any case, it is obvious that the information gained from the knowledge that a Standard ML program is properly typed will be essential in reasoning about that program.

SML Type	Corresponding *Val*s
`int`	"Val_Con n"
`'a * 'b`	"Val_Record $\{1 \mapsto L, 2 \mapsto R\}$"
`bool`	"Val_Con ‘true‘" "Val_Con ‘false‘"
`'a list`	"Val_Con ‘nil‘" "Val_Con_Val ‘cons‘ *mlpair*"

Table 1: Some Standard ML types and their *Val*-subsets

6.1 Concrete Types as Subsets of *Val*

The type *Val* which is defined the semantics encompasses all possible values that may be assigned to variables within Standard ML programs, as well as other, non-realizeable values. Each Standard ML type is associated with a subset of *Val* which realizes the type. In general, Standard ML *function* types such as `int -> int` are difficult to characterize in the context of the dynamic semantics. It would be helpful to be able to relate such a type to its subset of *Val*. However, any particular function type has an infinite number of corresponding possible closure-values to which it is semantically related. Furthermore, it is not possible to easily characterize this set, as this would involve embedding the entire typing system into HOL.

However, Standard ML *c*oncrete types such as `bool` and `'a list` do correspond to easily identified subsets of *Val*. It is with these types that we shall be concerned. A table of some simple types and their corresponding *Val*-subsets is shown in Table 1. It is necessary to deduce the properties of these subsets of *Val* to begin to reason about program-fragments using these types.

6.2 Using an Isomorphic HOL Type

The main technique used in this work to define the properties of a *Val*-subset is via reference to an isomorphic HOL type. For instance, the *Val*-subset for Standard ML lists is isomorphic to the HOL type `:list(Val)`. Similarly, the values for booleans are isomorphic to *bool*.

The *Val*-subset is defined by a predicate over the type *Val*. To relate this subset and its HOL type, bijections need to be defined between the two. An

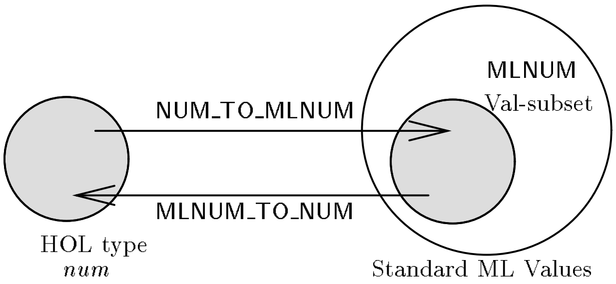

Figure 2: Relating a Standard ML Type to an Isomorphic HOL type

example of this for numbers is shown in Figure 2. The properties of the HOL type may then be "lifted" via the bijections to equivalent results over the *Val*-subset.

Automatic Definition and Theorem Lifting

By using the bijections it is quite easy to define operations over the *Val*-subset by reference to the corresponding HOL definition. For instance, addition over *Vals* is defined by:

```
⊢def m MLNUM_ADD m′ =
    NUM_TO_MLNUM(MLNUM_TO_NUM m + MLNUM_TO_NUM m′)
```

It is also possible to implement a proof technique which will automatically "lift" the properties of the HOL type to the *Val*-subset. This can automate the lifting of both definitions and theorems as shown below.

```
# lift_definition "n < m";;
⊢def n MLNUM_LT m =
      (MLNUM_TO_NUM n) < (MLNUM_TO_NUM m)

# LESS_TRANS;;
LESS_TRANS = ⊢ n < n′ ∧ n′ < n″ ⊃  n < n″

# lift_theorem LESS_TRANS;;
⊢ IS_MLNUM p ∧ IS_MLNUM q ∧ IS_MLNUM r ⊃
  p MLNUM_LT q ∧ q MLNUM_LT r ⊃
      p MLNUM_LT r)
```

This automated-lifting technique was a late innovation which could prove quite useful in other areas.

7 Verifying Standard ML Programs and Beyond

We are now in a position to apply the HOL theory of the Core language that have been developed to prove some illustrative results about Standard ML.

7.1 The Append Function

The first example to be considered is the verification of the correctness of a simple function - in particular the `append` function. The code for this function is shown below in the syntax of the Core.

```
val rec append = fn
   {1=nil, 2=M} => M |
   {1=cons {1=x,2=L},2= M} =>
     cons {1=x,2=append {1=L,2=M}};
```

It is most convenient to state the correctness of a function in terms of its match, M_{append}, shown underlined above. The `append` function is correct if and only if:

- The application of M_{append} in `append`'s local environment E_{append} to (v_1, v_2) evaluates to v_1 appended to v_2 for all list-values v_1 and v_2.
- There is no change of state due to the evaluation.

In HOL this goal is stated as:

```
"∀s v1 v2.
   IS_MLLIST v1 ∧ IS_MLLIST v2 ⊃
   EVAL (s, E_append, MLPAIR v1 v2) M_append =
      MLLIST_APPEND v1 v2, s"
```

Here MLLIST_APPEND is the HOL logical definition of appending two values, lifted by the techniques described in the previous section from the definition for APPEND.

The Proof in HOL

The proof in HOL follows by induction over v_1. Because we have assumed that v_1 is an `MLLIST`, this is very similar to an ordinary list induction.

The base case is the following goal:

```
"EVAL(s, E_append, MLPAIR MLLIST_NIL v2) M_append =
      MLLIST_APPEND MLLIST_NIL v2, s"
```

By calling `SML_EVAL_TAC` the goal can now be solved by symbolic evaluation. The evaluator proves that the call to `append` returns v_2 without a change in state, and the result follows easily:

$$
\begin{array}{rcll}
\mathcal{M}_{\scriptscriptstyle I}(\underline{\texttt{If}}\ b\ e_1\ e_2) & = & \mathcal{M}_{\scriptscriptstyle I}(e_1) & \text{if } \mathcal{M}_{\scriptscriptstyle B}(b) \\
 & & \mathcal{M}_{\scriptscriptstyle I}(e_2) & \text{if } \neg\mathcal{M}_{\scriptscriptstyle B}(b) \\
\mathcal{M}_{\scriptscriptstyle I}(\underline{\texttt{Num}}\ n) & = & n & \\
 & & & \\
\mathcal{M}_{\scriptscriptstyle B}(\underline{\texttt{Eq}}\ e_1\ e_2) & = & (\mathcal{M}_{\scriptscriptstyle I}(e_1) = \mathcal{M}_{\scriptscriptstyle I}(e_2)) & \\
\mathcal{M}_{\scriptscriptstyle B}(\underline{\texttt{Bool}}\ b) & = & b &
\end{array}
$$

Figure 3: The Denotational Semantics

```
# SML_EVAL_TAC THEN REWRITE_TAC [MLLIST_APPEND_NIL];;
goal proved...
```

The step case also follows by using symbolic evaluation and a simple theorem about MLLIST_APPEND. The induction hypotheses is employed to determine the result of the recursive call to `append` within the `append` function itself. With the proof of correctness in the step case the entire proof is completed.

Perhaps the most remarkable thing about the proof is that it makes almost no reference to the details of the semantics of the language. The symbolic evaluator performs all the reasoning steps about the language constructs that are needed. The evaluator has applied approximately 100 inference rules and many more minor results. The final form of the automated proof took about an hour to compose and a few minutes to execute, a proof by hand would take much longer.

7.2 Another Verification Result

One more verification proof that has been performed using the HOL theory of Standard ML will now be outlined. The function to be verified is a small interpreter over the language given by the `datatype` declarations below.

```
datatype IExp = If   of (BExp * IExp * IExp)
              | Num  of int
     and BExp = Eq   of (IExp * IExp)
              | Bool of bool
```

The interpreter itself is shown in Figure 4. The statement of correctness can be given in a manner similar to that for the `append` function, using the small denotational semantics shown in Figure 3 to supply the required meaning functions. The interpreter was deliberately written in a slightly convoluted style to demonstrate that the code need not look identical to the denotational semantics for the proof to work.

The full details of the proof shall not be shown here. Essentially it proceeds by structural induction over the different possible programs that may be input to the interpretation function. For each step and base case of the induction, symbolic evaluation is used over the appropriate constructs to prove that the correct result is produced. The induction hypotheses are employed to deduce the value of the sub-evaluations in each call.

```
val choose = fn (true,  (i1,_)) => i1 |
                (false, (_,i2)) => i2;

val rec interp_iexp =  fn
  If (bexp, (iexp1,iexp2)) =>
    let path =
      choose (interp_bexp bexp, (iexp1, iexp2)) in
    interp_iexp path  |
  Num n => n

and interp_bexp = fn
  Eq (iexp1,iexp2) =>
     interp_iexp iexp1 = interp_iexp iexp2 |
  Bool b => b;
```

Figure 4: The Interpreter

7.3 Computation Properties

We now move on from the topic of verification to briefly discuss how the semantics can be used to reason about the language in other ways. Termination can be defined by:

Definition 3 (Termination) *The evaluation of a phrase ph in a background consisting of s and E terminates if and only if there exists a result res and state s′ such that:*

$$s, E \vdash ph \Rightarrow res, s'$$

is inferable.

In HOL this is stated as:

$$
\begin{array}{l}
\vdash_{def}\ \texttt{TERMINATES}\ (s,\texttt{E})\ phrase\ = \\
\quad (\exists res\ s'.\ \texttt{INFERABLE}(s,E,phrase,res,s'))
\end{array}
$$

It is similarly easy to state what it means for a computation to cause no state changes and no exceptions:

$$
\begin{array}{l}
\vdash_{def}\ \texttt{NO\_SIDE\_EFFECTS}\ (s,E)\ phrase\ = \\
\quad \forall res\ s'. \\
\qquad \texttt{INFERABLE}(s,E,phrase,res,s') \supset\ (s = s'))
\end{array}
$$

$$
\begin{array}{l}
\vdash_{def}\ \texttt{NO\_EXCEPTIONS}\ (s,E)\ phrase\ = \\
\quad \forall res\ s'\ p. \\
\qquad \texttt{INFERABLE}(s,E,phrase,res,s') \supset\ \texttt{NOT\_PACK}\ res
\end{array}
$$

Via symbolic evaluation and the correctness theorem it is quite easy to prove that the `append` function satisfies all these conditions. These results also hold for the interpreter that has been verified. Similarly, it is possible to prove that the code

```
let val tmp = !store in
   (store := 6; store := tmp)
end;
```

causes no side effects, despite its use of imperative features.

7.4 The Associativity of `append`

The last result to be presented in this paper is a proof of the associativity property of the `append` function. This result is not a verification proof, for it asserts the semantic equivalence of two program fragments:

```
append (X, append(Y, Z)) ≡
append (append(X, Y), Z)
```

X, Y and Z are arbitrary expressions, which may be assumed to evaluate to some values *xval*, *yval* and *zval*. The evaluation of these expressions may also have side effects. Calling the two fragments P_1 and P_2, equivalence is stated by requiring that if P_1 evaluates to some value, then P_2 must evaluate to the same value, and vice-versa. The goal is stated in HOL as:

```
"IS_MLLIST xval ∧ IS_MLLIST yval ∧ IS_MLLIST zval ∧
 EVAL(s0, E0) append = Closure_append, s0 ∧
 EVAL(s0, E0) X = xval, s1 ∧
 EVAL(s1, E0) Y = yval, s2 ∧
 EVAL(s2, E0) Z = zval, s3 ⊃
  ((EVAL (s0, E0) P1 = res, s_final) =
   (EVAL (s0, E0) P2 = res, s_final))
```

The proof begins by applying the correctness result for the `append` function and an associativity result for HOL lists. This reduces the goal to the following subgoal:

```
"EVAL (s0, E0) P1 = res, s_final)
 EVAL (s0, E0) P2 = res, s_final)"
  [ EVAL (s0, E0) P1 =
     MLLIST_APPEND xval (MLLIST_APPEND yval zval), s2 ]
  [ EVAL (s0, E0) P2 =
     MLLIST_APPEND xval (MLLIST_APPEND yval zval), s2 ]
```

The two assumptions give the results of the evaluations for the left and right hand sides of the goal. The result now follows by applying the Determinacy Theorem. Note that without the Determinacy Theorem the result could *not* be concluded.

8 Summary

The aim of this work was to investigate aspects of machine assisted reasoning about the programming language Standard ML. The method used was to construct a HOL theory of the Core dynamic semantics of the language, based on its Formal Definition. A series of important results have been presented: the construction of an accurate embedding of the semantics into HOL; the development of a useful set of primitive rules from this basis; the proof of the Pattern Matching and Determinacy Theorems; the construction of a symbolic evaluator; the development of methods by which to reason with Standard ML concrete types in the context of the HOL theory; the proof of a substantial set of results including correctness proofs, termination proofs and the equivalence of program fragments. The work has demonstrated that the application of a tool such as HOL to such reasoning problems is quite feasible.

References

[1] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[2] Robert S. Boyer and J Strother Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, 1979.

[3] A. Cant and M.A. Ozols. A verification environment for ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 151–155, June 1992.

[4] Michael J. C. Gordon. Mechanizing programming languages in Higher Order Logic. *HOL System Documentation*, 1988.

[5] Michael J. C. Gordon et al. *The HOL System Description*, December 1989.

[6] HOL Users Group. HOL-INFO mailing list archives, 1992.

[7] Matthew Hutchins. Machine assisted reasoning about Standard ML using HOL. *Honours Thesis*, November 1990. Australian National University.

[8] Tom Melham. A package for inductive relation definitions in HOL. *Proceedings of the 1991 HOL Workshop*, 1991.

[9] Robin Milner. Lecture notes on language semantics. Computer Science 3, University of Edinburgh.

[10] Robin Milner. The Standard ML Core language. *Polymorphism*, 2(2), 1985.

[11] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1990.

[12] Rachel Onate-Ruxos. Mutually recursive inductive relation definitions in HOL. *HOL 1993 Workshop*, 1993. Preprint of submitted paper.

[13] Chris Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[14] Robin Milner Robert Harper and Mads Tofte. *The Definition Of Standard ML, Version 2*. MIT Press, 1990.

[15] Claudio V. Russo. Automating mutually recursive type definitions in HOL. *Honours Thesis*, June 1992. University Of Edinburgh.

[16] Donald Syme. Reasoning with the formal definition of Standard ML in HOL. *Honours Thesis*, November 1992. Australian National University.