Supporting Formal Reasoning about Standard ML

Graham Collins Stephen Gilmore

November 11, 1994

Abstract

A set of tools to support reasoning about Standard ML is described. These tools are based on the ML Kit compiler and the HOL-ML embedding of the dynamic semantics of Standard ML within the HOL90 interactive proof system. Examples are developed showing the use of these tools in reasoning about simple Standard ML program fragments.

1 Introduction

There are many practical difficulties involved in formal reasoning about a programming language. A large number of these are caused by the intricacy of the syntax and semantics of real languages which can make it impractical to reason with any great confidence about programs.

This report examines some of the difficulties that arise when trying to reason about programs in a particular language, Standard ML, and explores possible solutions. In order to carry out formal reasoning about a programming language it is essential to have a formal definition for the language. For Standard ML this exists in the form of the Definition of Standard ML [1] which defines the various syntactic and semantic objects which are the subject of much of this report.

One important feature of the Definition is that it is subdivided: the static semantics ensures a declaration is well-formed and well-typed while the dynamic semantics deals with evaluation. This subdivision has allowed proof systems to be built for formal reasoning purely about dynamic aspects of the language. One such proof system is HOL-ML [2], built on HOL90. We have composed this with the ML Kit compiler [3] to provide a system which allows the user a privileged insight into the workings of the Definition of Standard ML. The effect of the whole system is to behave as an interpreter but allowing additional information to be produced about the evaluation of terms and then subsequently supporting proofs of theorems about operational evaluation.

The structure of this paper: HOL-ML is described in the next section. The report then describes the ML Kit compiler and the changes made to the interface for this. In Section 4 we examine the problems caused by the complexity of the static and semantic objects. Section 5 discusses how the Kit compiler can

Email: grmc@dcs.gla.ac.uk, stg@dcs.ed.ac.uk

be further extended to support some automated proof and Section 6 gives some examples illustrating the use of the work in this report.

This paper is a summary of a dissertation [4] submitted for the degree of MSc at the University of Edinburgh.

2 HOL-ML

HOL-ML is an embedding of the dynamic semantics of *core* Standard ML, including exceptions and reference values but excluding module constructs (signatures, structures and functors). This embedding supports reasoning both about the core semantics of Standard ML and about programs of the core language. In keeping with the HOL tradition this embedding is purely definitional and uses no axiomatisation.

This is part of an ongoing project to encode all the semantics of Standard ML. Work on the dynamic semantics of the module system has already been completed by Savi Maharaj and Elsa Gunter [5]. Myra VanInwegen is currently working on the static semantics.

HOL-ML first defines types for the syntactic and semantic objects of Standard ML. Defining these within the system allows quantification over, for example, all expressions. HOL-ML then defines a family of evaluation relations, one for each syntactic class. The relation for expressions is `eval_exp`. The theorem

$$\vdash \texttt{eval\_exp}\ ex\ s_1\ e\ s_2\ vp$$

states that the expression ex can evaluate to vp in the environment e and state s_1, changing the state to s_2. A theorem of this form does not say that vp is the *only* value that ex can evaluate to, merely that it is a value allowed by the semantics. In fact Standard ML evaluation is deterministic and so such theorems about evaluation uniquely identify the result. The determinacy proof appears in the Commentary on Standard ML [6] and two separate machine-checked versions of this proof have been produced by Donald Syme [7] and Myra VanInwegen [8].

HOL allows both forward proof and backward, goal directed proof. Goal directed proof is supported by the subgoal package which allows the goal to be repeatedly decomposed into subgoals which can eventually be proved. This reduction is performed by tactics, modified by tacticals.

3 The ML Kit compiler

The ML Kit compiler animates the Definition as a Standard ML compiler implemented in Standard ML. The internal representation of objects such as the abstract syntax is very close to the Definition.

We used the Standard ML of New Jersey implementation of the Kit compiler, extending it to return the syntactic and semantic objects produced during Standard ML evaluation. In all the work below, the Kit compiler is used to keep track of the current state and environment. This has been achieved by modifying the Kit compiler so that these objects are not initialised each time an interactive session commences. Functions have been provided to allow the user to control the initialisation of semantic objects and the evaluation of declarations. These functions form a modified interface to the Kit compiler which is defined by the structure `Toolkit` and listed in Table 1.

`eval_dec`	Evaluates a Standard ML declaration in the current environment. Further calls to `eval_dec` are evaluated in the environment produced by this declaration as are calls to the other functions.
`elab_dec`	Similar to `eval_dec` but only elaborates the declaration and does not change the environment.
`eval`	Enters an interactive Kit session. It is possible to exit and re-enter the session or mix `eval` and `eval_dec` commands without losing the stored environment.
`undo`	Reverses the effect of one of the above commands by resetting the environment and state. It is only possible to undo once.
`reset`	Resets the semantic objects such as environments to their initial values which are described as the initial basis.
`clear`	Similar to `reset` but with the environment empty.

Table 1: The `Toolkit` functions

4 Syntactic and semantic representations

4.1 Syntactic objects

There are several problems related to parsing Standard ML which mean that it is not possible to parse the language using syntactic methods alone. It is necessary to also elaborate the syntax tree before all of the ambiguities can be fully resolved [9]. It is because of these complications that the resident parser generators of systems such as HOL cannot parse Standard ML code and require the code to be translated into a simpler form. Consider trying to translate the definition of a 'not' function to the HOL-ML version.

```
val not1 = fn true1 => false1 | false1 => true1
```

This fragment of code could be translated to two different abstract syntax trees depending on the context in which the elaboration occurs. Figure 1 shows the abstract syntax tree when `true1` and `false1` are declared as constructors. A different tree would result if `true1` and `false1` were values and the function would have a different meaning. In other programming languages constructors and values are lexically distinct but this is not the case in Standard ML.

We have now introduced a new step into the reasoning process and a new source of errors. It is clearly not possible to perform such a translation by hand for any but the simplest of examples if one wishes to have any reasonable degree of confidence in the correctness of the translation.

A tool to carry out this conversion has been implemented by using the Kit compiler. The syntax tree for any expression can be obtained by the function `get_ast` built from simpler functions, one for each syntactic class. Most of these functions omit some of the information stored by the Kit compiler (such as the source code line number). These functions are also used in the conversion of semantic objects. The abstract syntax tree can be converted to its HOL-ML version using the function `make_ast_holml`. This function raises an exception if the abstract syntax contains objects not in the subset of the Standard ML core language used by HOL-ML.

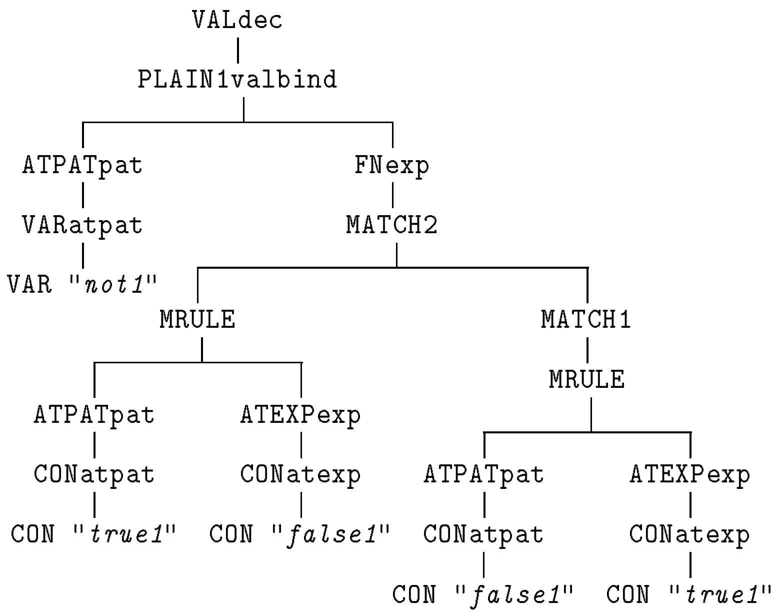

Figure 1: Syntax tree with `true1` and `false1` as type constructors

4.2 Semantic objects

The semantic objects provide a different set of problems. Here the problem is not just one of translation but of how to describe the objects at all. This section looks at ways to support the generation of these objects.

4.2.1 Environments and states

To prove something about evaluating a declaration in a particular environment and state, it is necessary to give that environment and state. These may be very complicated structures which can again be derived automatically by supplying the previous definitions and evaluating them to get the environment they create. Three functions are defined for this task. The functions `get_env` and `get_state` return the current environment or state at any point during a Kit compiler session. The function `get_change_env` returns the change in the environment produced by the last declaration to be evaluated.

As with the abstract syntax trees these functions are implemented in terms of functions which apply the conversion to each semantic object. Many of these functions simplify the types of semantic objects to allow easier manipulation. The Kit compiler represents many objects such as the variable environment by the type `FiniteMap`. These are converted to lists for ease of use later.

4.2.2 Values

For values the situation is rather more complex. In the formation of a goal it may be necessary to form the syntactic representation for a semantic object which is produced as the result of evaluating some fragment of code. If the code is a declaration then the result is simply the change in environment and state which can be accessed by the functions just described.

It may be desirable to be able to return the result of evaluating an object of any syntactic class. A problem arises because the top level of the Kit compiler evaluates declarations. This need not present a severe problem since it is possible to phrase other goals in terms of declarations and any change to the environment may be undone using `undo`.

4.2.3 Exceptions

In Standard ML the programmer can raise and handle exceptions. The 'result' of raising an exception is a *packet*: either an exception name or an exception name bound to an identifier. The exception name is some unique name assigned to the exception by the implementation and might bear no relation to the identifier for the exception constructor. The exception constructor identifier is not used during evaluation, it is for error reporting. In the Kit compiler exception names are implemented by references to the exception identifier and this guarantees that each exception name is unique. There is no way to convert these references directly to any useful representation so the system maintains an association list relating each exception name reference to a unique integer.

In the Kit compiler packets are represented by the exceptions of the implementation of Standard ML used to compile the compiler. In our case this is Standard ML of New Jersey. Raising a packet p is represented by raising the exception `EXCEPTION` p. In order to reason about packets it is necessary to trap these exceptions and the function `catch_pack` is used for this. The function takes an argument x of any type `'a` and returns a value of type `'a valpack` defined by the following datatype declaration.

```
datatype 'a valpack = VAL of 'a | PACK of Pack
```

The function `catch_pack` returns `VAL` x if x is not an exception packet and returns `PACK` p_1 if an exception of the form `EXCEPTION` p is raised where p_1 is a translation of p, using the association list described above.

5 Some automatic theorem proving

We have already seen how the Kit compiler can be used to calculate abstract syntax and maintain a record of the environment and state. The purpose for which a Standard ML interpreter exists is to evaluate declarations and the question that this raises is whether or not the mechanics of this evaluation can be of any use for proving theorems.

In this section we describe how the Kit compiler can be used to produce the derivation tree for the evaluation of a declaration and this is used to fully automate the proof that a declaration evaluates to its expected value. Of course this method will only work with fragments of Standard ML code and not with more general propositions such as those containing (implicitly or explicitly) quantified variables.

It is worth noting that this method will provide a *structured* proof, following the Definition precisely, rather than trying to solve a search problem or attempt the exhaustive application of rewrite rules.

The function `prove_dec` takes a Core ML declaration, in a subset of the language used by HOL-ML and containing no free variables, and returns a theorem about the evaluation of this declaration. A second function `prove_exp` performs a similar task for expressions. The subset of the language allowed includes many of the most commonly used constructs of Standard-ML, including exceptions and reference values, and could be extended to the full language used by HOL-ML.

```
datatype 'a except = OK of 'a | EXCEPT of exn

fun unpack (OK x) = x
  | unpack (EXCEPT x) = raise x

fun catch f x = OK (f (unpack x)) handle y => EXCEPT y
```

Figure 2: The `except` type constructor

5.1 Calculating the derivation tree

The derivation tree is produced by tracing the execution of the evaluation functions in the Kit compiler. This involved using a reference value to store the derivation and implementing 'wrapper' functions to trace each of the evaluation functions.

The handling of exception packets again caused some difficulty since they are represented by the exceptions of the underlying Standard ML compiler. It is necessary to both store these exceptions in the derivation tree and allow them to alter the course of the evaluation. The solution was to introduce a new type constructor, `except`, with some functions to trap all exceptions. Suppose we wish to apply a function `f` to a value `x`, storing the result of the evaluation even if it is an exception packet and then raising the exception. We can use the following code which will return the same value or exception as `f x` but can also update the state via an appropriately defined `store` function.

```
let val v = catch f (OK x)
in  (store v ; unpack v)
end
```

These functions produce a trace of the evaluation using the ML Kit compiler's internal objects. For each stage in the evaluation process we are storing the term evaluated, the semantic object produced and the new state together with the branches of the derivation tree at this point.

5.2 Theorem Proving

This section describes how a goal directed style of proof is used to prove the desired theorem. First, the method of forming the goal is described and then the method of deriving the tactic.

5.2.1 Stating the goal

For declarations each goal has the form

$$\texttt{eval\_dec}\ d\ s_1\ e\ s_2\ ep$$

where d is a declaration, s_1 and s_2 are the states before and after the evaluation of d, e is the environment in which d is evaluated and ep is the environment or packet produced by evaluating d. These may be constructed using the functions `get_ast`, `make_ast_holml`, `get_state`, `make_state_holml`, `get_env`, `make_env_holml` and `get_change_env`.

```
| prove (EXP (ex,_,_,l)) =
      (FIRST_ASSUM MATCH_MP_IMP_TAC) THEN
      (case ex of
         Ast_Grammar.ATEXPexp _ =>  ... |
         Ast_Grammar.APPexp _  =>   ... |
         Ast_Grammar.TYPEDexp _ =>  ... |
         Ast_Grammar.HANDLEexp _ => ... |
         Ast_Grammar.RAISEexp _ =>
               let val EXP (_,vp,_,_) = hd l
               in  (EXISTS_TAC
                     (parse_str (c_valpack vp))) THEN
                     CONJ_TAC  THENL
                     [(prove (hd l)),
                     (REWRITE_TAC [is_VALvp_DEF, EXVALval_arg_DEF,
                                   VALvp_arg_DEF])]
               end |
         Ast_Grammar.FNexp _ =>     ... |       )
```

Figure 3: Case analysis in the proof of expressions

In practice d, s_2 and ep will be calculated by looking at the root of the derivation tree produced from the Kit compiler and converting the necessary information using the conversion functions for declarations, states and environment-packets from the `Make_Dyn_HolML` structure.

5.2.2 Calculating the tactic

The tactic to solve the goal is formed by a case analysis over the type constructor for the current branch of the derivation tree. Each of these classes was then further divided according to the syntactic class being considered at that point. For example the function to generate a tactic for an `EXP` branch of the derivation tree has the form shown in Figure 3. The omitted code will form a tactic to prove the result for the part of the derivation tree being considered.

Suppose as part of a proof we needed to prove that

```
eval_exp (RAISEexp x1) s1 e1 s2 p1
```

where `x1` is an expression, `s1` and `s2` are the states before and after evaluation, `e1` is the environment in which evaluation takes place and `p1` is the exception packet produced. The corresponding node of the derivation tree will have the form `EXP(RAISEexp x,p,s,l)` where `x` is an expression, `p` is the exception packet to which `x` evaluates, `s` is the state after evaluation and `l` is the list of branches of the tree. The rule in HOL-ML which implements the rule for this case in the Definition is shown below.

$$
\begin{array}{l}
\forall s_1\ e\ s_2\ vp\ ex. \\
\quad (\exists vp'.\ \mathrm{eval\_exp}\ ex\ s_1\ e\ s_2\ vp' \wedge \\
\qquad (\mathrm{is\_VALvp}\ vp' \\
\qquad\quad \Rightarrow vp = \mathrm{PACKvp}\,(\mathrm{PACK}\,(\mathrm{EXVALval\_arg}\,(\mathrm{VALvp\_arg}\ vp'))) \\
\qquad\quad \mid \quad vp = vp')) \Longrightarrow \\
\qquad\qquad \mathrm{eval\_exp}\,(\mathrm{RAISEexp}\ ex)\ s_1\ e\ s_2\ vp
\end{array}
$$

This rule is in the assumptions of the goal and is applied by the tactic FIRST_ASSUM MATCH_MP_IMP_TAC which applies the first assumption that matches the goal. Applying the rule instantiates s_1, e, s_2, vp and ex to s1, e1, s2, p1 and x respectively. The goal at this point would be

$$
\begin{array}{l}
\exists vp'.\ \mathrm{eval\_exp}\ \mathtt{x1\ s1\ e1\ s2}\ vp' \wedge \\
\quad (\mathrm{is\_VALvp}\ vp' \\
\quad\quad \Rightarrow \mathtt{p1} = \mathrm{PACKvp}\,(\mathrm{PACK}\,(\mathrm{EXVALval\_arg}\,(\mathrm{VALvp\_arg}\ vp'))) \\
\quad\quad \mid\ \ \mathtt{p1} = vp')
\end{array}
$$

To determine the value for vp' all that is necessary is to look at the first derivation tree in the list l. This will be of the form EXP (ex,vp2,s,l). Converting vp2 to the HOL-ML form gives the required replacement for vp'. The replacement is done using EXISTS_TAC. The new goal is

$$
\begin{array}{l}
\mathrm{eval\_exp}\ \mathtt{x1\ s1\ e1\ s2\ vp2} \wedge \\
\quad (\mathrm{is\_VALvp}\ \mathtt{vp2} \\
\quad\quad \Rightarrow \mathtt{p1} = \mathrm{PACKvp}\,(\mathrm{PACK}\,(\mathrm{EXVALval\_arg}\,(\mathrm{VALvp\_arg}\ \mathtt{vp2}))) \\
\quad\quad \mid\ \ \mathtt{p1} = \mathtt{vp2})
\end{array}
$$

Applying CONJ_TAC splits this into two goals at the conjunction. The first goal is solved by a recursive call to the prove function, this time supplying the derivation from list l as the argument. There will only be one derivation in list l. The second goal is solved by using the tactic REWRITE_TAC [is_VALvp_DEF, VALvp_arg_DEF, EXVALval_arg_DEF] which rewrites with the definitions of is_VALvp, VALvp_arg and EXVALval_arg along with some internal rewrite rules which will rewrite the conditional $x \Rightarrow y \mid z$ construction. At this point there is nothing left to prove.

Joining these together with the THEN and THENL tacticals the final tactic to solve the goal is as shown in Figure 3. The FIRST_ASSUM MATCH_MP_IMP_TAC is before the case analysis because it is needed for every branch of the case statement.

Many of the other parts of the prove function are more complicated and often the witnesses for the existentially quantified variables are not obvious.

The tactic is used to prove the goal using the function TAC_PROOF which takes the goal and tactic and applies the tactic to the goal.

6 Examples

This section discusses two examples of the uses of the tools described. The examples here are small and are shown either with the print depth set to reduce the amount of information printed or with the initial environment set to a very small subset of the full initial basis. This is necessary to reduce the size of the environments which rapidly become very large.

Larger examples have been attempted including examples illustrating reference values and examples illustrating the definition and use of a sorting function.

6.1 The not function

The first example attempted was not. This is not trivial since the proof requires a significant subset of the rules for the language to be automated. This included closures and matches in order to implement function application. The proof for

```
|- eval_exp
     (APPexp (ATEXPexp (VARatexp (VAR "not")))
             (CONatexp (CON "false")))
     (STATE (MEM (ADDR 0) NONEmemmap) (EXNAMESET (EXNAME 5)))
     (ENV
        (SOMEvarenv (VAR "not")
           (CLOSUREval
             (CLOSURE
                (MATCH2
                   (MRULE (ATPATpat (CONatpat (CON "true")))
                     (ATEXPexp (CONatexp (CON "false"))))
                  (MATCH1
                    (MRULE (ATPATpat (CONatpat (CON "false")))
                      (ATEXPexp (CONatexp (CON "true"))))))
                (ENV NONEvarenv NONEexconenv)
               NONEvarenv))
          NONEvarenv)
       NONEexconenv)
     (STATE (MEM (ADDR 0) NONEmemmap) (EXNAMESET (EXNAME 5)))
     (VALvp (CONval (CON "true"))) : thm
```

Figure 4: Theorem for `not false`

`not false = true` is given in Figure 4. The complicated syntax given there is the formalisation of

$$E,\ s \vdash \texttt{not false} \ \Rightarrow \texttt{true},\ s'$$

With `not` defined as

```
val not = fn true => false | false => true
```

Once the system was able to prove this case it was also able to prove the evaluation for other expressions such as `not (not false)` with no modification. It is also possible to use the toolkit to get a more general result about applying `not` twice. We would like to be able to prove the following theorem.

$$\frac{E,\ s \vdash x \Rightarrow v,\ s'}{E,\ s \vdash \texttt{not (not } x\texttt{)} \Rightarrow v,\ s'} \tag{1}$$

The first difficulty is that this cannot be proved from the dynamic semantics alone because x could evaluate to a value which is not a boolean and then `not (not` x`)` would evaluate to an exception. Using an assertion that x evaluates to `true` or `false` we can prove the following theorem.

$$\frac{E,\ s \vdash x \Rightarrow v,\ s' \quad (v = \texttt{true}) \vee (v = \texttt{false})}{E,\ s \vdash \texttt{not (not } x\texttt{)} \Rightarrow v,\ s'} \tag{2}$$

Theorem (2) can be proved for a specific state and environment using `prove_exp1` to generate the tactics that solve the following goals.

$$E,\ s \vdash \texttt{not (not true)} \Rightarrow \texttt{true},\ s \tag{3}$$

$$E,\ s \vdash \texttt{not (not false)} \Rightarrow \texttt{false},\ s \tag{4}$$

```
- eval_dec "exception X";
> exception X
[end of <string>]
val it = () : unit

- prove_exp "not (raise X)";
val it =
  |- eval_exp
       (APPexp (ATEXPexp (VARatexp (VAR "not")))
         (PARatexp (RAISEexp (ATEXPexp (EXCONatexp (EXCON "X"))))))
       (STATE (MEM (ADDR 0) NONEmemmap) (EXNAMESET (EXNAME 6)))
       (ENV
          (SOMEvarenv (VAR "=") (BASval Eql)
              (SOMEvarenv (VAR "not") (CLOSUREval ( ... ))
                (SOMEvarenv ( ...   ... ) (BASval Less)
                  ( ...   ...  ( ...   ... ) NONEvarenv))))
         (SOMEexconenv (EXCON "X") (EXNAME 1)
           (SOMEexconenv (EXCON "Sum") (EXNAME 0) NONEexconenv)))
       (STATE (MEM (ADDR 0) NONEmemmap) (EXNAMESET (EXNAME 6)))
       (PACKvp (PACK (NAMEexval (EXNAME 1)))) : thm
```

Figure 5: Declaration and proof about evaluation of an exception

Using `STRIP_TAC` on (2) and rewriting with the assumptions reduces the goal to a `true` case and a `false` case. Applying the tactics that prove (3) and (4) to these subgoals completes most of the proof. The result follows by some rewriting using the assumptions. Below are the proof steps that the user must supply.

```
(REPEAT STRIP_TAC) THEN
(ASM_REWRITE_TAC[]) THENL
[(snd ct) THEN
 (FIRST_ASSUM MATCH_MP_IMP_TAC) THEN
 (SUBST_TAC
     [ONCE_REWRITE_RULE [EQ_SYM_EQ]
         (ASSUME (--`y = VALvp (CONval (CON "true"))`--))]) THEN
(ASM_REWRITE_TAC[]),
 (snd cf) THEN
 (FIRST_ASSUM MATCH_MP_IMP_TAC) THEN
 (SUBST_TAC
     [ONCE_REWRITE_RULE [EQ_SYM_EQ]
         (ASSUME (--`y = VALvp (CONval (CON "false"))`--))]) THEN
 (ASM_REWRITE_TAC[])]
```

The variable *ct* is the result of `prove_exp1 "not (not true)"` and the variable *cf* is the result of `prove_exp1 "not (not false)"`. The second elements of these pairs are the tactics to solve (3) and (4).

6.2 Examples using exceptions

An example session is given in Figure 5 which shows the declaration of an exception `X` and the result of applying `not` to an expression which raises that exception. The last example illustrated how important the structure of the derivation tree is. The derivation tree that aids the proof in Figure 5 will be of an identical structure for any environment, any state and any function application where the evaluation of the argument raises an exception.

$$
\begin{array}{l}
\forall\, f\ x\ e\ s\ s_1\ s_2\ v\ p. \\
\quad \text{eval\_exp}\, f\ s\ e\ s_1\ (\texttt{VALvp}\, v) \wedge \\
\quad\quad \text{eval\_exp}\, x\ s_1\ e\ s_2\ (\texttt{PACKvp}\, p) \Longrightarrow \\
\quad\quad\quad \text{eval\_exp}\, (\texttt{APPexp}\, f\ (\texttt{PARatexp}\, x))\ s\ e\ s_2\ (\texttt{PACKvp}\, p)
\end{array}
$$

Because the structure of the derivation tree will be the same for all such evaluations and the proof tool develops the proof only on the basis of the derivation tree the structure of the proof generated exactly matches the general case.

To test this it is simply necessary to modify the code of the proof tool so that instead of instantiating the state and environment labels to those for a specific case it instantiates them to HOL variables. The point at which the instantiations occur is not difficult to find. They occur in the `APPexp` branch of the tool. Only four lines need to be modified and the proof tool then generates a proof for the general case.

A few basic tactics have to be used to tidy up the use of the assumptions in the goal. If the tactic to solve the example in Figure 5 is t then the goal is proved by this tactic.

```
(REPEAT STRIP_TAC) THEN
t THENL
[(ASM_REWRITE_TAC[]),
 (FIRST_ASSUM MATCH_MP_IMP_TAC) THEN
 (ASM_REWRITE_TAC[])]
```

7 Conclusion

This report describes some tools which simplify the manipulation of the syntactic and semantic objects of the Definition of Standard ML. The work described in Section 4 provides the required tools for converting Standard ML code into the largely unreadable syntax required by the HOL theorem prover. By using the Kit compiler to do much of the work and keep track of all the details the system places no additional restrictions on the user. The system does not force the user to distinguish between various classes of identifier nor does it prohibit the use of derived forms. It lets the user enter pieces of real Standard ML code.

None of the work on extending the Kit compiler is dependent on the HOL-ML system. This work could equally well be used as a foundation for a toolkit for any other system where it is necessary to convert Standard ML concrete syntax to abstract syntax for input into the system. The tool for producing the derivation tree is also independent of HOL-ML and could also be used as an aid to proof in other theorem proving systems.

Only tools for supporting the syntax and dynamic semantics of Core Standard ML are given here. Tools for converting the objects for the static semantics would not be significantly different from those for the dynamic semantics. Similarly tools for supporting reasoning about the module system could also be easily implemented.

Acknowledgements

This work was supported by an Engineering and Phyical Sciences Research Council studentship.

References

[1] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[2] Myra VanInwegen and Elsa Gunter. HOL-ML. In J. J. Joyce and C. J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 61 –74. Spinger-Verlag, 1993.

[3] Lars Birkdal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit—Version 1. March 1993.

[4] Graham Collins. Supporting Formal Reasoning about Standard ML. Master's thesis, The University of Edinburgh, 1994.

[5] Savi Maharaj and Elsa Gunter. Studying the ML Module System in HOL. In Tom Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *LNCS*. Springer-Verlag, September 1994.

[6] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[7] Donald Syme. Reasoning with the Formal Definition of Standard ML in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1993.

[8] Myra VanInwegen. The formal specification of programming languages. PhD Thesis Proposal, University of Pennsylvania, September 1994.

[9] Nick Haines. ML and parsing—a position paper. In Peter Lee, editor, *ACM Sigplan Workshop on ML and its Applications, San Fransisco, California.*, June 1992. also published as Carnegie Mellon University technical report CMU-CS-93-105.