Studying the ML Module System in HOL

Elsa Gunter[1] and Savi Maharaj[2]

[1] AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, N.J. 07974, USA. email: elsa@research.att.com

[2] LFCS, University of Edinburgh, Edinburgh EH9 3JZ, UK. email: svm@dcs.ed.ac.uk

In an earlier project [8] the dynamic semantics of the Core of Standard ML (SML) was encoded in the HOL theorem-prover. We extend this by adding the dynamic Module system. We then develop a possible dynamic semantics for a Module system with higher-order functors and encode this as well. Next we relate these two semantics via embeddings and projections and discuss how we can use these to state and to prove that evaluation in the proposed system is a conservative extension, in an appropriate sense, of evaluation in the SML Module system.

1. INTRODUCTION

This paper describes an application of the HOL theorem-prover [2] to the problem of reasoning about the semantics of a programming language. The language in question is Standard ML (SML), which consists of a strongly-typed, higher-order, functional programming language (with some imperative constructs), plus a system of modules which allow structuring and information hiding. We shall refer to these two parts as the Core and the Modules, respectively. SML is particularly suitable for our purpose because it has been given a concise formal semantics [5]. This semantics is presented in a structured operational semantics style [6] which makes it relatively easy to reason about it mathematically and to translate it into a mathematical formalism such as the language of HOL. The size of SML — for example the number of different phrase classes in its grammar, and the number of inference rules in its operational semantics — means that there is a great advantage to be gained in the use of a theorem prover for automating the tedious inductive proofs of the properties of the language.

The semantics of SML is given in *The Definition of Standard ML* [5], which we shall refer to as the *Definition*. It is divided into four parts, giving the *static* and then the *dynamic* semantics of first the Core and then the Modules. Roughly speaking, static semantics concerns well-typedness and certain well-formedness constraints for modules, while dynamic semantics concerns the evaluation of terms under the assumption that static properties have already been checked.

We make use of earlier work [8] by Myra VanInwegen and Elsa Gunter who encoded the dynamic semantics of Core SML in HOL and then formally verified the fact that dynamic evaluation is deterministic. (A similar project carried out by Donald Syme [7] could conceivably have been used instead.) The encoding is done via definitions and derived properties, as opposed to using an axiomatisation. This approach necessitates the use of a theorem prover with a higher-order language, but was chosen because it provides induction principles — powerful tools for proving meta-theoretic properties of the semantics as a whole. (Instead of only being able to prove consequences of individual inference rules, we can also show properties such as determinism that concern the totality of inference rules.) These can then be used to prove facts about evaluation that could not be proved otherwise. Examples of these are negative facts such as that certain expressions do not evaluate. Our approach also has the advantage that it automatically preserves the consistency of the logic, i.e. it does not introduce the possibility of proving a false statement, whereas the axiomatic approach requires special principles for the particular extensions desired.

In our project we build upon the previous work by adding the dynamic semantics of the SML Module system. We then propose a speculative dynamic semantics for an extension of SML with *higher-order functors*. A functor, in SML, is a parameterised program module, whose parameter is a non-parameterised module called a *structure*. The point of the extension is to allow functors to take other functors as argument. Work has been done on the (considerably more complicated) static semantics of this extension [1] but, to the best of our knowledge, this is the first time a dynamic semantics has been written down. By presenting this semantics within a theorem prover we gain assistance in proving theorems that give confidence in the proposed semantics. Such theorems include that the extended Module system is, in a sense which we shall define, a conservative extension of the original system. To clarify the presentation in this paper we shall use a simplified version of SML in which we omit, among other things, the constructs for dealing with persistent state and with exceptions for error handling. However, in the formal encoding we treat the full language.

In Section 2. we describe the fragments of SML syntax and semantics concerned with dynamic evaluation of Modules, and discuss the way in which we represent these in HOL. In Section 3. we present our proposed extended Module system and discuss various issues that arise in deciding upon a semantics for this system. We then describe the HOL encoding of the extended system. In Section 4. we discuss how to state and prove that the proposed system is a conservative extension of Modules. Finally we give our concluding remarks in Section 5..

2. MOD-ML

The constructs which make up the SML Module system are called *structures*, *signatures*, and *functors*. A structure is a code module: it encapsulates a sequence of declarations of variables or other structures. The identifiers declared inside a structure can be referred to from outside by use of *long identifiers*, e.g. `S.x`. Figure 1. shows a small structure `S` which contains a variable `x` and an structure `SS`. Signatures are *specifications* or *views* of structures. New structures can be formed by using signatures to form smaller versions (i.e. having fewer components) of existing structures, a process we call *thinning*. In the figure, S is thinned by the signature SIG to produce a smaller structure T containing only `x`. Functors are parameterised structures, whose parameter is specified by a signature. The output of a functor may optionally be constrained by a signature, as in Figure 1. Functors may be applied to structures to form new structures.

Syntax

The dynamic semantics uses a reduced syntax for SML in which information that is not relevant to evaluation has been removed. The abstract syntax classes for Modules include those for the Core and are shown in Fig. 2. Structures (*strexp*,$\cdots$,*strbind*) can contain Core declarations and structure declarations and can be built up from other structures and functors (through application) in various ways. Signatures (*valdesc*, $\cdots$, *sigdec*) can specify values and structures and can be formed in several ways, including extracting the signatures from a list of structures (OPENspec). Functors (*funbind*, *fundec*) are defined by binding a functor identifier to a structure identifier and signature, which together specify the input to the functor, an optional output signature (In the figure, optional syntactic elements are enclosed within angle brackets.), and a structure expression which makes up the functor body.

Semantic objects The *Definition* gives the meanings of programs of the Module system in terms of *semantic objects* which we show in Fig. 3. The notation Fin(X) and X $\stackrel{fin}{\rightarrow}$ Y is used for finite subsets and finite functions, respectively. Top declarations are evaluated with respect to a basis, which contains information about previously declared modules. The result of the evaluation is itself a new basis. Structure declarations evaluate to give environments (Env) which contain information about the variables and structures contained in the declared structure. Coincidentally, these environments are the same as those used in Core evaluation (the structure information is needed to evaluate long identifiers), so the Core semantic object can be re-used. Signature bodies (Spec) are evaluated to interfaces (Int) which contain information needed to later thin structures to fit the signature. This includes the names of variables and the internal interface environment. Functors are evaluated to give a functor closure consisting of an identifier and interface for the functor argument, an interface for thinning the output, if supplied, the structure expression that comprises the functor body, and a copy of the basis in which applications of the functor are to be evaluated.

Functions on the semantic objects The semantics makes use of various auxiliary functions which operate upon the semantic objects. These include functions for looking up identifiers in the various environments and bases, written as e.g. $G(sigid)$; projecting the components of semantic objects, denoted by e.g. E of B; injecting objects as components of larger objects, written as e.g. E in Basis, which denotes the basis $(\{\},\{\},E)$; updating environments and bases, written as e.g. $G + \{sigid \mapsto I\}$. Functions are lifted in the obvious way to operate on larger objects which contain the domain of the function as a component. For example, one can look up signature identifiers in bases (by looking them up in the SigEnv component).

The semantics of OPENspec, which allows a signature to be formed by extracting the signatures from a list of structures, demands a special function to perform this extraction. The structures are evaluated to produce environments, and then from each environment we must extract an interface. This is done by the function Inter : Env $\rightarrow$ Int defined as:

$$\text{Inter}\ (SE, VE) = (IE,\ \text{domain}\ VE)$$

where

$$\text{IE} = \{strid \mapsto \text{Inter}\ E\ ;\ SE\ (strid) = E\}.$$

That is, for structure environments, the interface we extract is the set of mappings from the *strid*s they contain to interfaces for the *env*s they associate with the *strid*s. For variable environments the extracted interfaces are just the sets of identifiers they contain.

Another important function, $\downarrow$: Env $\times$ Int $\rightarrow$ Env, thins an environment to fit a given interface. This is needed whenever a signature is explicitly given to a structure. It is defined by:

$$(SE, VE)\ \downarrow\ (IE, vars) = (SE', VE')$$

where

```
structure S =  struct val x = 6  structure SS = struct end end
signature SIG = sig val x : int end
structure T:SIG = S
functor F (Arg:SIG):SIG = struct val x = Arg.x end
structure U = F(S)
```

FIGURE 1. SML Module system examples

var	::=	VAR *string*	variable (Core)
dec	::=	(omitted)	declaration (Core)
strid	::=	STRID *string*	structure identifier (Core)
longvar	::=	$strid_1.\cdots.strid_n.var \quad (n \geq 0)$	long variable (Core)
longstrid	::=	$strid_1.\cdots.strid_n.strid \quad (n \geq 0)$	long structure identifier (Core)
sigid	::=	SIGID *string*	signature identifier
funid	::=	FUNID *string*	functor identifier
strexp	::=	STRUCTstrexp *strdec* $\mid$	structure expression
		LONGSTRIDstrexp *longstrid* $\mid$	
		APPstrexp *funid strexp* $\mid$	
		LETstrexp *strdec strexp*	
strdec	::=	DECstrdec *dec* $\mid$	structure declaration
		STRUCTUREstrdec *strbind*	
		LOCALstrdec *strdec strdec* $\mid$	
		EMPTYstrdec $\mid$	
		SEQstrdec *strdec strdec*	
strbind	::=	BINDstrbind *strid* $\langle$*sigexp*$\rangle$ *strexp* $\langle$*strbind*$\rangle$	structure binding
valdesc	::=	VARvaldesc *var*	value description
strdesc	::=	STRIDstrdesc *strid sigexp*	structure description
spec	::=	VALspec *valdesc* $\mid$	specification
		STRUCTUREspec *strdesc* $\mid$	
		LOCALspec *spec spec* $\mid$	
		OPENspec $longstrid_1 \cdots longstrid_n$ $(n \geq 1)$ $\mid$	
		INCLUDEspec $sigid_1 \cdots sigid_n$ $(n \geq 1)$ $\mid$	
		EMPTYspec $\mid$ SEQspec *spec spec*	
sigexp	::=	SIGsigexp *spec* $\mid$ SIGIDsigexp *sigid*	signature expression
sigbind	::=	BINDsigbind *sigid sigexp* $\langle$*sigbind*$\rangle$	signature binding
sigdec	::=	SIGNATUREsigdec *sigbind* $\mid$	signature declaration
		EMPTYsigdec $\mid$	
		SEQsigdec *sigdec sigdec*	
funbind	::=	BINDfunbind *funid strid sigexp* $\langle$*sigexp*$\rangle$ strexp $\langle$*funbind*$\rangle$	functor binding
fundec	::=	FUNCTORfundec *funbind* $\mid$	functor declaration
		EMPTYfundec	
		SEQfundec *fundec fundec*	
top	::=	STRDEC *strdec* $\mid$ SIGDEC *sigdec* $\mid$ FUNDEC *fundec*	top declaration

FIGURE 2. Modules abstract syntax

$E \in$	$\mathrm{Env} = \mathrm{StrEnv} \times \mathrm{VarEnv}$	environment (Core)
$SE \in$	$\mathrm{StrEnv} = strid \xrightarrow{fin} \mathrm{Env}$	structure env. (Core)
$VE \in$	$\mathrm{VarEnv} = var \xrightarrow{fin} \mathrm{Val}$	value environment (Core)
$v \in$	$\mathrm{Val} = (\mathrm{omitted})$	value (Core)
$I \in$	$\mathrm{Int} = \mathrm{IntEnv} \times \mathrm{Fin}(var)$	interface
$IE \in$	$\mathrm{IntEnv} = strid \xrightarrow{fin} \mathrm{Int}$	interface environment
$G \in$	$\mathrm{SigEnv} = sigid \xrightarrow{fin} \mathrm{Int}$	signature environment
$IB \in$	$\mathrm{IntBasis} = \mathrm{SigEnv} \times \mathrm{IntEnv}$	interface basis
$FC \in$	$\mathrm{FunctorClosure} = (strid \times \mathrm{Int}) \times (strexp\ \langle \times\ \mathrm{Int}\rangle) \times \mathrm{Basis}$	functor closure
$FE \in$	$\mathrm{FunEnv} = funid \xrightarrow{fin} \mathrm{FunctorClosure}$	functor environment
$B \in$	$\mathrm{Basis} = \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env}$	basis

FIGURE 3. Modules semantic objects

$$SE' = \{strid \mapsto E \downarrow I\ ;\ SE(strid) = E \text{ and } IE(strid) = I\}$$

and VE' is formed by restricting the domain of VE to *vars*.

The evaluation relations The main part of the semantics consists of inference rules that define evaluation relations by which the phrase classes are related to the semantic objects. We shall not give these in detail here.

Encoding the Semantics in HOL

We encode the semantics of the Module system in HOL and call the resulting package of definitions Mod-ML. We shall write the names of Mod-ML types and terms in `teletype` font. Mod-ML is an extension of the system HOL-ML [8] which is an encoding of the dynamic semantics of a large subset of the Core language of SML. The whole of HOL-ML must be loaded into HOL in order for Mod-ML to be loaded.

Syntax The logic supported by the HOL system contains an extensible system of types. We added new types to this collection to represent each of the Modules phrase classes. First we separated out the phrase classes into groups whose members happen to be mutually recursive with each other. By defining these groups separately so as to avoid unnecessary mutual recursion among the types we were able to simplify the presentation of our definitions, theorems and inductive proofs. Next, to make the type definitions, we used a package developed by Elsa Gunter and Healfdene Goguen which supports simultaneously mutually recursive and *nested* recursive definitions of types. An example of the latter is a pair of mutually recursive types, S and T, say, where one of the constructors of S takes as an argument a value of type (T list). This package both extends and depends upon a previous package for mutual recursion developed by Elsa Gunter and Myra VanInwegen and first used in the development of HOL-ML.

We re-use various techniques developed for HOL-ML: optional arguments to constructors in the grammar are encoded by defining a polymorphic type `'a option` (here `'a` is a HOL type variable) with the constructors `NONE` and `SOME 'a`; the lists of identifiers used by OPENspec and INCLUDEspec are represented by defining a type `'a nonemptylist`. For long identifiers we define a new type `'a long`.

Semantic objects and functions To encode the semantic objects we must make choices about the representation of finite sets and finite function spaces. The HOL system is well supplied with libraries containing definitions and theories about useful mathematical objects, including sets, lists, and pairs. To represent finite sets we use the library's (possibly infinite) sets, since we are free to add finiteness later as a hypothesis to theorems which require it. For finite functions we use lists of pairs of the appropriate identifier and value types. When we encode functions that are intended to operate on finite function spaces we are always careful to ensure that the lists are maintained in lexicographical order by identifiers. Our aim here is to make the list structure transparent so that we adequately represent finite functions.

Evaluation To complete the encoding we define the evaluation relations which say how syntactic terms evaluate to the appropriate semantic object. For those phrase classes that form individual recursive types we use the HOL command `new_inductive_definition` to define the evaluation relation. This command defines a relation from a family of rules that give an inductive description of the relation. Unfortunately it is capable of defining only a single relation, not a mutually recursive family of such relations. Therefore for phrase classes that form a nested or mutually recursive group, and thus have evaluation relations defined by mutual recursion, we must do the corresponding job by hand. For example, the phrase classes *strexp*, *strdec* and *strbind* form a mutually recursive group. To define their evaluation relations, we first define an evaluation-relation predicate for the group. This predicate, `ModML_eval_structures_pred` is defined over possible evaluation relations for the three phrase classes, and is true if the possible evaluation relations satisfy all the rules for evaluating the three phrase classes. Then we define each evaluation relation as the logical intersection of all relations that satisfy the evaluation-relation predicate. For example, the evaluation relation for *strexp* is defined as (simplified):

```
eval_strexp strexp:strexp B:basis E:env =
  ∀ e_strexp e_strdec e_strbind.
    ModML_eval_structures_pred
            e_strexp e_strdec e_strbind ⇒
      e_strexp strexp B E
```

Defining the evaluation relations in this manner has the advantage that it gives us an induction principle for proving facts about them. The last thing we do is to prove that each evaluation relation satisfies the appropriate evaluation-relation predicate. HOL provides us with a number of built-in *tactics* (SML functions which operate on proof goals) for doing proofs and *tacticals* for composing tactics into SML functions which embody powerful proof strategies. To prove the above property, and also to derive the induction principles, we used two tactics which were a concise composition of built-in tactics parameterised by the definitions.

strexp	::=	(as in Fig. 1) | APPstrexp *longfunid strexp* | (as in Fig. 1)
moddec	::=	DECmoddec *dec* | STRUCTUREmoddec *strbind* |
		LOCALmoddec *moddec moddec* |
		OPENmoddec $longstrid_1 \cdots longstrid_n$ $(n \geq 1)$ |
		EMPTYmoddec | SEQmoddec *moddec moddec* |
		FUNCTORmoddec *funbind*
spec	::=	(as in Fig. 1) | FUNCTORspec *funid strid sigexp sigexp*
funbind	::=	(as in Fig. 1) | REBINDfunbind *funid longfunid*
top	::=	MODDEC *moddec* | SIGDEC *sigdec*

FIGURE 4. Abstract syntax for higher-order functors (additions and changes)

3. HIGHER ORDER FUNCTORS

It has been proposed (Section 8.5 of [4], [1]) to extend SML by allowing functors to take functors as arguments and to be declared within structures (and therefore to be specified in signatures). A possible static semantics [1] has been outlined for these "higher-order" functors. Here we use HOL to work out what the dynamic semantics of this extension should be, and then to explore the relationship between the extended system and the original system. For readability we present the new semantics in the informal notation used in the *Definition*, but we should like to stress that this semantics was developed *within* HOL.

Syntax In deciding upon a syntax for the extended language, we have been motivated by the desire to make it easy to define the relationship between the extension and the original language. We made no changes to the Core language. Changes to the grammar of the Module system are listed in Fig. 3 and explained here.

Structure expressions (*strexp*) We now have long *funid*s, referring to functors declared within structures, and we can apply these to form new structures.

Module declarations (*moddec*) Functor declarations are now to be treated as a special kind of structure declaration, so to support this we amalgamate these two phrase classes into a new class of module declarations. There exists a Core SML declaration by which the variable and structure declarations within a structure can be exposed to the top level. When we extend to higher-order functors we choose to keep this syntax with its original semantics — which means that it does not expose the functors within a structure. To allow these to be exposed we add a module level declaration, OPENmoddec which exposes all variable, structure, and functor bindings.

Specifications (*spec*) Functors can now be specified in signatures. This is done by giving a *funid* for the whole functor, a *strid* and signature for the input structure, and a signature for the output structure.

Functor bindings (*funbind*) It seems to be an omission in the SML grammar that no syntax is supplied for rebinding a functor to another functor identifier. We remedy this since we think this is a useful language feature as it can be used to give top-level names to functors declared within structures, and to rebind functors passed through a functor's parameter.

Top declarations (*top*) The change here reflects the fact that structure declarations and functor declarations have been combined.

Semantic objects

The main difficulty in deciding upon semantic objects is determining what environments should be. As has been shown, environments play a dual role in the dynamic semantics of the SML Module system. They give the values associated with long identifiers during the evaluation of Core expressions, and they are themselves the values returned by the evaluation of structure expressions. SML structures contain only structure declarations and Core declarations, and it happens that this is precisely the information needed for Core evaluation. However, once we allow functors within structure bodies, the situation changes. The environment returned by evaluating a structure now must contain information about the functors declared within the structure. Core evaluation has no use for this new information. Therefore, we are faced with two alternatives: either use environments in the Core semantics which have excess information, or define two kinds of environment, one for Core evaluation (the one we already have), and one for structure values. We decided to take the second option in this paper, but in future work we intend to encode both approaches and prove that they are essentially the same.

The choice to have two different kinds of environment has some ramifications. One of these is the need, which we have mentioned before, to have two different kinds of `open` declarations: one which throws away functor information (for the Core language) and one which exposes it (for the Module system). Another consequence is that we must define how to cut a Module-level environment down to a Core-level environment to enable the passing of evaluation between the Module system and the Core. We shall use the notation E of ME for this function. Its definition is straightforward. Another straight-forward function is the lifting of a Core environment to a Module environment, denoted by E in ModEnv.

The semantic objects for higher-order functors are those defined in Fig. 5, plus the classes SigEnv, FunctorClosure and FunEnv which remain unchanged from Fig. 3, plus the Core semantic objects.

- **Interfaces** (Int) Interfaces are prescriptions for how to thin the view of a structure. Since structures may now contain functors, interfaces must now prescribe how to thin the view of a functor. Therefore they contain a new component: a functor interface environment.
- **Structure Interface Environments** (StrIntEnv) These are the equivalent of interface environments in SML. We have renamed them to reflect their function in the semantics of higher-order functors.
- **Functor Interface Environments** (FunIntEnv) These contain interface information to be used in thinning functors. The nature of this information is discussed at length later.
- **Interface Bases** (IntBasis) These now have a new component: a functor interface environment.
- **Module-level environments** (ModEnv) These are the environments obtained as the result of evaluating structures. Since structures can contain functors, these environments contain a functor environment (FunEnv) component. In the rest of this paper we will refer to these objects as "environments" unless there is a possibility of confusion with Core-level environments.
- **Module-level structure environments** (ModStrEnv) These are the Module-level counterparts of the Core-level structure environments (StrEnv).
- **Bases** (Basis) Bases no longer need to contain a separate functor environment since this has been moved into the ModEnv component.

Functions on semantic objects

Most of the projection, injection, and modification functions on the new semantic objects can be defined by straightforward changes to the corresponding functions in the SML semantics. Here we describe those functions that are significantly different:

Extracting interfaces Interfaces and environments now contain information about functors, so we must change the definition of Inter which extracts an interface from an environment. The new definition is as follows:

$$\text{Inter}\ (FE,\ MSE,\ VE) = (FIE,\ SIE,\ \text{domain}\ VE)$$

where

$$FIE = \{funid \mapsto \text{Inter\_funclos}\ FC\ ;\ FE\ (funid) = FC\}$$

and, as before,

$$SIE = \{strid \mapsto \text{Inter}\ ME\ ;\ MSE\ (strid) = ME\}$$

This is fine, except that we have not defined Inter_funclos yet. Inter_funclos is a significant complication that arises in the dynamic semantics of the higher-order Module system but is not present in the original system. Inter_funclos extracts interface information from a functor closure. For reasons we will explain later, the only interface information that is required for a functor is the interface of its output structure. If the functor closure is explicitly constrained (i.e., it arose from a functor that was provided with an output signature), then Inter_funclos returns the supplied output interface. However, if the original functor was unconstrained, then an output interface must be calculated from the structure expression that forms the body of the functor. This means we must define how to extract an interface from an arbitrary structure expression. Now we are on a slippery slope, because structure expressions can contain every other category in the grammar, except top declarations. Therefore, we must define the contribution of each grammatical category (except top declarations) to interfaces. Making these definitions is long and tedious, and we omit any further discussion of how it is done. It is worth commenting that using automated assistance to type-check the terms in our definitions and to warn us of any cases we had missed did speed the process of making the definitions and increased our confidence that we had made them correctly.

This complication does not arise in the SML modules system because environments there do not contain functor closures, and hence do not contain structure expressions.

Let us reflect for a moment on which feature of the language necessitates the function Inter and all the other interface-extraction functions it requires. An interface is the semantic equivalent of a signature expression. An environment is the semantic equivalent of a structure expression. So when do we syntactically express the act of turning a structure into a signature? This occurs when we open a structure within a signature (*viz.* OPENspec). This is intended to add the signature of the structure to the signature containing the OPENspec. One might reasonably ask if this is a desirable language feature. However, this language feature is clearly present in the *Definition*, so we felt we would not be carrying out the task of extending the specification if we chose to omit it.

Thinning environments Interfaces become more complicated in the setting of higher-order Modules because they must contain information concerning how to thin the view of a functor. In Fig. 5 we defined interfaces but did not explain how we decided what their functor components should be. Here we explain how we arrived at our choice.

Functor closures are thinned in a manner prescribed by functor specifications (i.e. *spec*s of the form FUNC-

$$
\begin{array}{rl}
I \in & \mathrm{Int} = \mathrm{FunIntEnv} \times \mathrm{StrIntEnv} \times \mathrm{Fin}(\mathit{var}) \\
\mathit{SIE} \in & \mathrm{StrIntEnv} = \mathit{strid} \xrightarrow{fin} \mathrm{Int} \\
\mathit{FIE} \in & \mathrm{FunIntEnv} = \mathit{funid} \xrightarrow{fin} \mathrm{Int} \\
\mathit{IB} \in & \mathrm{IntBasis} = \mathrm{FunIntEnv} \times \mathrm{SigEnv} \times \mathrm{StrIntEnv} \\
\mathit{ME} \in & \mathrm{ModEnv} = \mathrm{FunEnv} \times \mathrm{ModStrEnv} \times \mathrm{VarEnv} \\
\mathit{MSE} \in & \mathrm{ModStrEnv} = \mathit{strid} \xrightarrow{fin} \mathrm{ModEnv} \\
B \in & \mathrm{Basis} = \mathrm{SigEnv} \times \mathrm{ModEnv}
\end{array}
$$

FIGURE 5. Semantic objects for higher-order functors (additions and changes)

TORspec *funid strid sigexp sigexp*). These provide us with two interfaces (signatures): one describing the input taken by a functor and another describing the structure produced by a functor. There are three possibilities for using these interfaces in thinning a functor closure.

The first possibility is to replace the first interface of the functor closure by the (larger) first interface of the functor specification. This has the effect of guaranteeing that the functor body will receive a larger environment with more bindings from its input structure. This means that when the functor body is evaluated, more values will be looked up in the input environment. There is no change to the bindings available in the resulting output environment. The second possibility is to replace the second interface of the functor closure by the (smaller) second interface of the functor specification. Functors thinned in this manner will take exactly the same inputs as they did before thinning. However, when such functors are applied, the resulting environment will have fewer components than those produced by applying the unthinned versions. The third possibility is to combine both the first and the second definitions of thinning.

We believe that the second definition of thinning is the correct one. The first method of thinning (and consequently also the third) can result in the wrong environment being used for final computations. Consider the example below, in which we use a possible concrete syntax to improve readability.

```
val x = 5
functor F(I:sig end) =
   struct open I val z = x end
signature SIG =
   sig functor F(I:sig val x : int end)
              : sig val z:int end
   end
structure A = struct functor F = F end
structure B:SIG = A
structure I = struct val x = 6 end
structure A1 = A.F(I)
structure A2 = B.F(I)
val test = (A1.z = A2.z)
```

If we use the first method of thinning, we find that `A1.z` = `5` and `A2.z` = `6`, when it should be the case that `A1.z` = `5` = `A2.z`. For computing `z`, `F` requires the `x` in the top-level environment be used, and this should remain the case if we subsequently thin `F`. Thinning should change only the visibility of identifiers, not the underlying computations, and hence not the environments used for identifier lookup. With the second form of thinning both `A1.z` and `A2.z` would have the value `5`, as expected.

We therefore chose to record in the functor interface only the second (i.e. output) interface provided by a functor specification, and to thin functor closures by replacing only their output interfaces. Here is the definition of how to thin an environment:

$$
\begin{aligned}
&(\mathit{FE},\ \mathit{MSE},\ \mathit{VE}) \downarrow (\mathit{FIE},\ \mathit{SIE},\ \mathit{vars}) = \\
&\qquad (\mathit{FE}',\ \mathit{MSE}',\ \mathit{VE}')
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{FE}' = \{\ & \mathit{funid} \mapsto ((\mathit{strid},\ I),\ (\mathit{strexp},\ I_2),\ B)\ ; \\
& \mathit{FE}\,(\mathit{funid}) = ((\mathit{strid},\ I),\ (\mathit{strexp},\ I_1),\ B) \ \text{ and} \\
& \mathit{FIE}\,(\mathit{funid}) = I_2\ \}
\end{aligned}
$$

and, as in SML,

$$
\begin{aligned}
\mathit{MSE}' = \{ \mathit{strid} \mapsto \mathit{ME}{\downarrow}I\ ;\ & \mathit{MSE}\,(\mathit{strid}) = \mathit{ME} \text{ and} \\
& \mathit{SIE}\,(\mathit{strid}) = I\ \}
\end{aligned}
$$

and VE' is obtained by restricting the domains of VE to *vars*.

Evaluation In general we obtain the evaluation rules for the new language by modifying the rules of SML to work with the new semantic objects and functions in the obvious way. Many of the changes are trivial and we shall not describe them here. We must also add new rules to deal with the syntax we have added and make significant changes to some other rules. We list these in Fig. 6 and explain their meanings here, giving pointers to relevant rules in the *Definition* for readers who have access to a copy.

1. This rule defines how to evaluate the application of a *longfunid* to a *strexp*. [Replaces rule 162.]
2. This rule shows how to evaluate a Core declaration in a basis B: we must first extract a Core environment from B, use this to evaluate the declaration, and then lift any resulting Core environment to a Module environment to get the final result. [Replaces rule 164.]

3. This rule gives the semantics of the Module-level `open` declaration. [Insert after rule 166.]
4. This rule shows how to evaluate a functor declaration, now that functor declarations are specific instances of module declarations. [Insert after rule 168.]
5. This rule describes how to evaluate a functor specification. [Insert after rule 183].
6. This rule gives the semantics of rebinding a functor to a new identifier. [Insert after rule 187.]

Encoding higher-order functors We encode the new semantics into HOL by the same techniques used to encode the semantics of SML Modules. The encoding is called HOF-ML. Some of the new phrase classes and semantic objects are identical to the old ones, so for these we simply re-use the types used to encode these in Mod-ML. Some types have to be redefined despite being apparently unchanged, because they happen to be in mutual recursion with a type that has been changed. We distinguish the names of the HOF-ML constructors and types that have been redefined by appending `_h` to them. Thus, for example, the type representing HOF-ML structure descriptions is `strdesc_h` with constructor `STRIDstrdesc_h`. We do this because we will eventually want to have both Mod-ML and HOF-ML present in HOL together so that we can prove theorems about the relationship between them.

4. RELATING THE TWO SEMANTICS

So far we have described the encoding of two possible Module systems to extend the Core language of SML. But what is the rigorous connection between them? It is our claim that the system specified by HOF-ML is a conservative extension of the system specified by Mod-ML (and the *Definition*).

Before we can discuss how to prove such a result, we must state exactly what we are trying to prove. Recall that top declarations are evaluated within a basis to produce another basis. We want to prove that there is a function `embed_top` mapping top declarations of Mod-ML into top declarations of HOF-ML, a function `embed_basis` mapping Mod-ML bases into HOF-ML bases, and a function `proj_basis_h` mapping HOF-ML bases back to Mod-ML bases, such that both of the following conditions hold:

- For every top declaration *top* and pair of bases B_1 and B_2 of Mod-ML,

  $$eval\_top\ \ top\ \ B_1\ \ B_2$$

  holds if and only if

  $$eval\_top\_h\ \ (embed\_top\ \ top)\ \ (embed\_basis\ \ B_1)\ \ (embed\_basis\ \ B_2)$$

  also holds.

- For each basis B and top declaration `top` of Mod-ML, and basis B_h of HOF-ML, if

  $$eval\_top\_h\ \ (embed\_top\ \ top)\ \ (embed\_basis\ \ B)\ \ B_h$$

  holds, then

  $$eval\_top\ \ top\ \ B\ \ (proj\_basis\_h\ \ B_h)$$

  also holds.

Informally, the first condition says that an evaluation performed in Mod-ML is still valid when translated into HOF-ML. The second condition states that to evaluate a top declaration of Mod-ML, it suffices to translate into HOF-ML, evaluate there and translate the result back. This statement of conservative extension focuses on top declarations and bases. However, just to define the functions `embed_top`, `embed_basis`, and `proj_basis_h`, we need to define the corresponding functions for all categories of syntax and semantics in Mod-ML and HOF-ML.

Embedding Mod-ML in HOF-ML

Embedding the syntax of Mod-ML into that of HOF-ML is generally straightforward. Some phrase classes, such as identifiers are embedded by the identity function since they are represented by the same HOL types in both Mod-ML and HOF-ML. We give the flavour of the embedding by showing three of the clauses for Mod-ML structure expressions, declarations and bindings:

```
embed_strexp (STRUCTstrexp strdec) =
            STRUCTstrexp_h (embed_strdec strdec)
embed_strdec (STRUCTUREstrdec strbind) =
        STRUCTUREmoddec_h (embed_strbind strbind)
embed_strbind (BINDstrbind strid strexp) =
        BINDstrbind_h strid (embed_strexp strexp)
```

The only clause whose embedding is not trivial is `APPstrexp`. There the functor identifier that is applied must be lifted to a long functor identifier. Both functor and structure declarations are mapped to the appropriate kinds of HOF-ML Module declarations. Similarly, top-level functor declarations must be mapped to top-level Module declarations. The embeddings are trivial for all other cases. Defining an embedding of the semantic objects of Mod-ML into those of HOF-ML is also easy.

Projecting HOF-ML back to Mod-ML

It might appear that we only need to project the semantic objects of HOF-ML into Mod-ML, and can forget about the syntax, since the conservativity result only uses the projection of semantic objects (bases, to be precise). Unfortunately, this is not so. To project bases we need to project functor environments, and hence functor closures. To project functor closures we need to project structure expressions — syntax. With the exception of this dependency, the definition of the projection functions for the semantic objects is straightforward. The

$\boxed{B \vdash strexp \Rightarrow ME/p}$

$$\frac{\begin{array}{c} B\ (longfunid) = (strid : I, strexp'\langle : I'\rangle, B') \\ B \vdash strexp \Rightarrow ME_1 \quad B' + \{strid \mapsto ME_1{\downarrow}I\} \vdash strexp' \Rightarrow ME_2 \end{array}}{B \vdash longfunid(strexp) \Rightarrow ME_2\langle{\downarrow}I'\rangle} \tag{1}$$

$\boxed{B \vdash moddec \Rightarrow ME/p}$

$$\frac{E \text{ of } (ME \text{ of } B) \vdash dec \Rightarrow E'}{B \vdash dec \Rightarrow E' \text{ in Modenv}} \tag{2}$$

$\boxed{B \vdash moddec \Rightarrow ME/p}$

$$\frac{B(longstrid_1) = ME_1 \ \cdots \ B(longstrid_n) = ME_n}{B \vdash \texttt{open}\ longstrid_1 \ \cdots \ longstrid_n \Rightarrow ME_1 + \cdots + ME_n} \tag{3}$$

$\boxed{B \vdash moddec \Rightarrow ME/p}$

$$\frac{B \vdash funbind \Rightarrow FE}{B \vdash \texttt{functor}\ funbind \Rightarrow FE \text{ in ModEnv}} \tag{4}$$

$\boxed{IB \vdash spec \Rightarrow I}$

$$\frac{IB \vdash sigexp \Rightarrow I_1 \quad IB + \{strid \mapsto I_1\} \vdash sigexp' \Rightarrow I_2}{IB \vdash \texttt{functor}\ funid(strid\!: sigexp) : sigexp' \Rightarrow \{funid \mapsto I_2\} \text{ in Int}} \tag{5}$$

$\boxed{B \vdash funbind \Rightarrow FE}$

$$\frac{B\ (longfunid) = funclos}{B \vdash funid \ \texttt{=}\ longfunid \Rightarrow \{funid \mapsto funclos\}} \tag{6}$$

FIGURE 6. Evaluation rules for higher-order functors

only complication is that when we project a basis we must first pull the environment it contains into its constituent parts to access the functor environment and the structure environment and project them to acquire the corresponding components of a basis in Mod-ML.

We do not need to project all HOF-ML syntax back into Mod-ML; we only need those syntax classes that can be involved in bases. This simplifies the process somewhat. In particular, we never actually have to project functor declarations back to Mod-ML, because any functor declaration occurring in an HOF-ML basis came from a functor declaration internal to a structure expression, and therefore cannot correspond to anything in Mod-ML. It is fortunate that we may ignore functor declarations, for if we had to project them, we would have to deal with the fact that embeddings of sequences of empty functor declarations are indistinguishable from embeddings of sequences of empty structure declarations. This problem does not arise for us. All projection functions that we actually need have straightforward definitions.

Throughout the definitions of the embedding and projection functions, just as with the functions for extracting interfaces, we relied heavily on the package for nested mutually recursive types, and its support for generating definitions for functions from mutually recursive specifications over those types.

Proving Conservativity

Although the result relating the evaluation of top declarations in the two Module systems mentioned above is our main statement of conservative extension, in order to prove it we need to prove corresponding results for all layers of the evaluation relations. In the formal proof, we simplified the process by first showing the corresponding results for signature expressions, descriptions, and specifications, and then working up through the syntax classes.

We shall informally describe the proof of the result for top declarations. To further simplify the process, we show each of the two parts of the main conservativity theorem separately. Moreover, we split the first condition into the two halves of the equivalence, and we use the second condition to show the second half of the first condition.

The forward half of the first condition follows in a straightforward manner from the induction principles for the evaluation rules in Mod-ML and from the rules themselves, once we have proved a large number of easy lemmas stating that the embedding functions commute with the semantic operations such as identifier lookup and function update.

To prove the second condition, we must first coerce it into a form suitable for use with the induction principles for evaluation in HOF-ML. For example, the second condition for the syntax class `top` becomes:

$$\begin{array}{l} \forall\, top\_h\; B_h\; B'_h .\; eval\_top\_h\;\; top\_h\;\; B_h\;\; B'_h \Rightarrow \\ \quad \forall\, top\; B.\; ((top\_h = embed\_top\;\; top)\; \wedge \\ \qquad\qquad (B_h = embed\_basis\;\; B)) \Rightarrow \\ \quad\; eval\_top\;\; top\;\; B\;\; (proj\_basis\_h\;\; B'_h) \end{array}$$

Once we perform this transformation, we can apply the induction principle to reduce the problem to showing that the conclusion of the resulting implication holds for all the evaluation rules. Unfortunately, this result does not follow from induction; we need to prove a stronger result. In order to show the previous result, it turns out that we need to know that the result of embedding the projected value is the same as the original value before projection. This is not true in general, but happens to be true of the results of evaluations of expressions embedded from Mod-ML. This fact must be added to the conclusion of the second condition before we can proceed by induction. Thus, the second condition is transformed to

$$\begin{array}{l} \forall\, top\_h\; B_h\; B'_h .\; eval\_top\_h\;\; top\_h\;\; B_h\;\; B'_h \Rightarrow \\ \quad \forall\, top\; B.\; ((top\_h = embed\_top\;\; top)\; \wedge \\ \qquad\qquad (B_h = embed\_basis\;\; B)) \Rightarrow \\ \qquad (embed\_basis\; (proj\_basis\_h\;\; B'_h) = B'_h)\; \wedge \\ \qquad (eval\_top\;\; top\;\; B\;\; (proj\_basis\_h\;\; B'_h)) \end{array}$$

It was non-trivial to prove that embedding the result of projecting an embedded expression yields the original expression.

Finally, to prove the second half of the first condition from the second condition, it suffices to show that, if we embed a basis and then project, we end up where we started. It is too much to expect that such a result would hold for the entirety of Mod-ML, since empty structure declarations and empty functor declarations both get mapped to empty module declarations. However, the result holds for all the syntax and semantic classes involved in the range of the projection functions used in the statement of conservativity.

To show all the results discussed above, we have various tools at our disposal, including structural induction and case analysis over both the syntax and the semantics; rewriting with theorems that state the distinctness of all the constructors; rewriting with the equations that give the recursive "definitions" of the embedding and projection functions, the functions for extracting interfaces, *etc.* Moreover, by proving the results in a bottom-up fashion, starting with the earliest syntax classes, we have the results for this classes at our disposal when proving the later results.

While there is a great deal of regularity involved in carrying out the proofs of the different layers of the first half of the first condition, it is not apparent at present that we could write a general-purpose tactic that would automatically prove all of them. Each case seems to have just enough that is distinct about it to require interactive guidance. The second condition has a fairly involved proof and we can see no way to develop a general-purpose tactic to prove results of that kind.

5. CONCLUSION

We have described how we used the HOL theorem prover to specify the dynamic aspects of a higher-order Module system for SML, and then to relate it to the SML Module system specification. It is our belief that this task is too large to be easily managed by hand, the *Definition* notwithstanding. Using the expressiveness of HOL, the packages built into it, and packages we added to it, we were able to formulate the specification with the theorem prover as fast or possibly faster than we could do it by hand. Moreover, we have received some assurances that our specification makes sense from the type-checking of the terms, the checks that no clauses were omitted from our function definitions, and other checks that were performed automatically by HOL. Most importantly, by encoding the specification in a theorem prover, we are now able to formally prove facts about the specification and about programs written in complying implementations. Not only did we receive benefits from the theorem prover, but the theorem prover also received benefits from us. The specification task has motivated us to improve HOL's handling of mutually recursive types, and to write a general-purpose package for defining mutually recursive families of relations and deriving the appropriate induction principles. All of these benefits were made possible by the combination in HOL of an expressive language in which much general mathematics can be developed, with an open yet secure system which allows users to develop theorem-proving methodologies to suit their particular needs.

REFERENCES

[1] MacQueen, D.B. and Tofte, M. (1994) A Semantics for Higher-Order Functors. In: *European Symposium on Programming, 1994.* Springer-Verlag.

[2] Gordon, M.J.C. and Melham, T. (1993) *Introduction to HOL.* Cambridge University Press.

[3] Melham, T.F. (1992) A Package for Inductive Relation Definitions in HOL. In: *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications* IEEE Computer Society Press. Pages 350-357.

[4] Milner, R. and Tofte, M. (1991) *Commentary on Standard ML.* The MIT Press, Cambridge, Mass.

[5] Milner, R. and Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* The MIT Press, Cambridge, Mass.

[6] Plotkin, G.D., (1981) *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Computer Science Department, Århus University.

[7] Syme, D. (1994) Reasoning with the Formal Definition of Standard ML in HOL. In: *Higher Order Logic Theorem Proving and its Applications, 1993.* Lecture Notes in Computer Science 780, Springer-Verlag.

[8] VanInwegen, M. and Gunter, E. (1994) HOL-ML. *ibid.*