Studying the ML Module System in HOL

Savi Maharaj*[1] and Elsa Gunter[2]

[1] University of Edinburgh, Department of Computer Science
JCMB, King's Buildings, Edinburgh EH9 3JZ.
phone: 44 31 650 5183 email: svm@dcs.ed.ac.uk
[2] AT&T Bell Laboratories, Rm.#2A-432
600 Mountain Ave., Murray Hill, N.J. 07974, USA
phone: 1 908 582 5613 email: elsa@research.att.com

Abstract. In an earlier project [5] 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

The project described in this paper is part of an ongoing program of work on encoding the semantics of the SML programming language within HOL in order to support formal reasoning about the semantics and about SML programs. The encoding is done via definitions and derived properties, as opposed to using an axiomatization. This approach necessitates the use of a higher-order language, but was chosen because it provides us with induction principles, powerful tools for proving meta-theoretic properties of the semantics and facts about evaluation that follow from such properties. Examples of these are negative facts such as that certain expressions do not evaluate.

In earlier work [5] Myra VanInwegen and Elsa Gunter encoded the dynamic semantics of Core SML and then formally verified the fact that dynamic evaluation is deterministic. In our project we build upon this foundation by adding the dynamic semantics of the SML Module system (Modules). We then propose a speculative dynamic semantics for a Module system with higher-order functors for an extension of SML. To the best of our knowledge this is the first time that such a 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.

* Partially supported by AT&T

In Section 2 we first describe the fragments of SML syntax and semantics concerned with dynamic evaluation of Modules, and then discuss the way in which we represent these in HOL. In Section 3 we describe our proposed extended Module system with higher-order functors and discuss various issues that arise in deciding upon a semantics for this system. We also describe the HOL encoding of the extended system. In Section 4 we discuss in detail 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 semantics of SML is presented in *The Definition of Standard ML* [4], 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 the satisfaction of sharing requirements, while dynamic semantics concerns the evaluation of terms under the assumption that static properties have already been checked.

Syntax

The dynamic semantics makes use of a reduced syntax for SML in which information that is not relevant to dynamic evaluation has been removed. The syntax classes for Modules consist of Core identifiers and declarations, extended by signatures, structures and functors. The abstract syntax for Modules is shown in Fig. 1 along with a small part of the Core abstract syntax. Signatures (*valdesc*, $\cdots$, *sigdec*) are allowed to specify values, exceptions and structures and can be formed in several ways, including extracting the signatures from a list of structures (OPENspec). Structures (*strexp*,$\cdots$,*strbind*) can contain Core declarations and structure declarations and can be built up from other structures in various ways. Functors (*funbind, fundec*) are defined by binding a functor identifier to a structure identifier and signature, specifying the input to the functor, an optional output signature, 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 present in Fig. 2. We use the notation Fin(X) and $\mathrm{X} \stackrel{\mathrm{fin}}{\to} \mathrm{Y}$ to represent finite subsets and finite functions, respectively. Module declarations are evaluated with respect to a basis, which contains information about previously declared modules. 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 exceptions and the internal interface environment. Structure declarations evaluate to give environments (Env) which contain information about the variables, exceptions 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

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

Fig. 1. Modules abstract syntax

can be re-used. Functor declarations are evaluated to give a FunEnv mapping the functor identifier to a functor closure consisting of an identifier and interface for the input structure, an interface for the output, if supplied, the body of the functor, and 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 looking up identifiers in the various environments and bases, denoted by e.g. *G (sigid)*,

$E \in \text{Env} = \text{StrEnv} \times \text{VarEnv} \times \text{ExConEnv}$		environment (Core)
$SE \in \text{StrEnv} = \mathit{strid} \xrightarrow{\text{fin}} \text{Env}$		structure env. (Core)
$VE \in \text{VarEnv} = \mathit{var} \xrightarrow{\text{fin}} \text{Val}$		value environment (Core)
$EE \in \text{ExConEnv} = \mathit{excon} \xrightarrow{\text{fin}} \text{ExName}$		excon environment (Core)
$\text{Val} = \text{(omitted)}$		value (Core)
$\text{ExName} = \text{(omitted)}$		exception name (Core)
$I \in \text{Int} = \text{IntEnv} \times \text{Fin}(\mathit{var}) \times \text{Fin}(\mathit{excon})$		interface
$IE \in \text{IntEnv} = \mathit{strid} \xrightarrow{\text{fin}} \text{Int}$		interface environment
$G \in \text{SigEnv} = \mathit{sigid} \xrightarrow{\text{fin}} \text{Int}$		signature environment
$IB \in \text{IntBasis} = \text{SigEnv} \times \text{Intenv}$		interface basis
$\text{FunctorClosure} = (\mathit{strid} \times \text{Int}) \times (\text{StrExp} \langle \times \text{Int} \rangle) \times \text{Basis}$		functor closure
$F \in \text{FunEnv} = \mathit{funid} \xrightarrow{\text{fin}} \text{FunctorClosure}$		functor environment
$B \in \text{Basis} = \text{FunEnv} \times \text{SigEnv} \times \text{Env}$		basis

Fig. 2. Modules semantic objects

which means look up *sigid* in the signature environment G; 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. $G + \{\mathit{sigid} \mapsto I\}$. 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:

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

where

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

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 and exception 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:

$$(SE, VE, EE) \downarrow (IE, \mathit{vars}, \mathit{excons}) = (SE', VE', EE')$$

where

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

and VE' and EE' are obtained by restricting the domains of VE and EE to *vars* and *excons* respectively.

The evaluation relations The main part of the semantics consists of inference rules giving the 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 described in [5], which is an encoding of the dynamic semantics of a large subset of the Core language of SML. All of the definitions that make up HOL-ML must be loaded into HOL in order for Mod-ML to be loaded. In both encodings, SML syntax and semantic objects are represented in HOL either as new inductive types or by means of objects (such as sets and lists) already present in the HOL libraries. Evaluation relations, relating the terms of each syntactic class to the corresponding semantic objects, are defined inductively as the smallest relations satisfying various evaluation-relation predicates representing the rules of the dynamic semantics.

Syntax Each of the phrase classes in Fig. 1 is represented by a type in HOL. Rather than introducing all of these types at once as one huge mutually recursive type definition, we have chosen to separate them out into mutually dependent groups and define each of these separately. We believe that by avoiding unnecessary mutual recursion among the types the presentation of definitions and theorems will be made clearer and inductive proofs will be simpler. To make these definitions we used a package developed by Elsa Gunter and Healfdene Goguen which supports simultaneously mutually recursive and nested recursive definitions of types. 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 in [5]: optional arguments to constructors in the grammar are encoded by defining a type `'a option` with 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. For the former we use the sets supplied by the HOL library since finiteness can be added later as an axiom to theorems which require it. For the latter 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 ordering 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. Again we re-use the techniques of [5]. 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 giving 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 inductive 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 inductive 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 =
  ∀ poss_eval_strexp poss_eval_strdec poss_eval_strbind.
    ModML_eval_structures_pred
            poss_eval_strexp poss_eval_strdec poss_eval_strbind ⇒
      poss_eval_strexp strexp B E
```

Defining the evaluation relations in this manner has the advantage that it readily 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. This and the proof of the induction principles were done by two tactics which were a concise composition of built-in tactics parameterized by the definitions.

3 Higher order functors

It has been proposed (Section 8.5 of [3], [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). No definitive semantics has yet been proposed for these “higher-order functors” though a possible static semantics is outlined in [1]. 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 for the SML semantics, but we should like to stress that this semantics was developed *within* HOL.

Syntax A possible syntax for higher-order functors is given in [1]. However this differs from SML syntax in various idiosyncratic ways ([1] is a speculative, draft paper) which obfuscate the relationship between terms in the two languages. Therefore we decided to develop our own grammar by starting with the grammar of SML and making changes for the new constructs. The grammar for the Core language remains unchanged. Changes to the grammar of the Module system are listed in Fig. 3 and explained here.

$$
\begin{array}{lll}
\textit{spec} & ::= & \text{(as in Fig. 1)} \mid \text{FUNCTORspec } \textit{funid strid sigexp sigexp} \\
\textit{strexp} & ::= & \text{(as in Fig. 1)} \mid \text{APPstrexp } \textit{longfunid strexp} \mid \text{(as in Fig. 1)} \\
\textit{moddec} & ::= & \text{DECmoddec } \textit{dec} \mid \text{STRUCTUREmoddec } \textit{strbind} \mid \\
 & & \text{LOCALmoddec } \textit{moddec moddec} \mid \\
 & & \text{OPENmoddec } \textit{longstrid}_1 \cdots \textit{longstrid}_n \mid \qquad (n \geq 1) \\
 & & \text{EMPTYmoddec} \mid \text{SEQmoddec } \textit{moddec moddec} \mid \\
 & & \text{FUNCTORmoddec } \textit{funbind} \\
\textit{funbind} & ::= & \text{(as in Fig. 1)} \mid \text{REBINDfunbind } \textit{funid longfunid} \\
\textit{topdec} & ::= & \text{MODDEC } \textit{moddec} \mid \text{SIGDEC } \textit{sigdec}
\end{array}
$$

Fig. 3. Abstract syntax for higher-order functors (additions and changes)

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

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 the two phrase classes into a new class of module declarations.

Core SML provides a feature (`open`) by which the 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 functor bindings within a structure. To allow functor bindings to be exposed we add a version of `open` (OPENmoddec) to the language of module declarations.

Functor bindings (*funbind*) It appears 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 this language feature is important to us as we can use it to give top-level names to functors declared within structures and to rebind functors passed through a functor's parameter.

Top declarations (*topdec*) 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 figuring out what environments should be. As we have seen, environments play a double role in the dynamic semantics of the SML Module system. They tell us the values associated with long identifiers during the evaluation of a Core expression, and they are the values returned by the evaluation of structure expressions. Since SML structures contain only structure declarations and Core language declarations, the environments they generate contain precisely the information needed by Core evaluation. However once we allow functor bindings within structure bodies, the situation changes. The environments needed for Core evaluation require only enough information to allow long identifiers to be looked up; they need no information concerning functors and functor bindings. However the environments returned by structures now do need to contain information concerning the functors bound in the body of the structure. Therefore, we are faced with two alternatives: either use environments in the Core dynamic semantics which have excess information, or define two kinds of environment, one for Core evaluation being the one we already have, and one for structure values. In the *Definition* there are already two kinds of "environments": environments for Core evaluation and bases for Module evaluation. We therefore decided that the second option was most in keeping with the spirit of the *Definition*, and this is the choice we have pursued here. In future work, we intend to encode both approaches and prove that the two are essentially the same.

The choice to have two different kinds of environments has ramifications elsewhere. One of these is the need 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 ramification is that we are obliged to 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.

The semantic objects for higher-order functors are those defined in Fig. 4, plus the classes SigEnv, FunctorClosure and FunEnv which remain unchanged from Fig. 2.

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

Fig. 4. Semantic objects for higher-order functors (additions and changes)

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 more accurately 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, which now can contain functors. To reflect this, 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 SML functions. 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:

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

where

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

and, as before,

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

This is fine, except that we haven't defined Inter_funclos yet. Inter_funclos is a significant complication that arises in the dynamic semantics of the higher-order Module system that is not present in the original system, since interfaces there did not need to make mention of functors. The interface information we have decided to keep for functors is the interface of the output structure. We discuss why this is the right choice (as opposed to using the interface for the input structure, or both, or neither) in the next subsection. If the functor closure is constrained (by an interface arising from an original constraining signature), then

Inter_funclos extracts the interface constraining the output structure. However, if the functor closure is unconstrained, then we must calculate the interface from the structure expression describing the output structure of the functor. That is, we must be able to extract an interface from syntax. Now we are on a slippery slope, because structure expressions can and do contain every other category in the grammar, except top declarations. Therefore, we have to define the contribution of each grammatical category (except top declarations) to interfaces. Making these definitions is long and rather tedious, and we omit any further discussion of how it is done here. 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 have made them correctly.

Let us reflect for a moment on which feature of the language necessitates the function Inter for extracting interfaces from environments, 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 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 `open`. One might reasonably ask if this is a desirable language feature. However, this language feature is clearly present in the *Definition*, and we felt we would not be carrying out the task of extending the specification if we simply 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. 3 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.[3] These provide us with two interfaces (signatures) : one describing the input taken by a functor and another describing the structures 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 applied, the resulting environments will have fewer components than those produced by applying the unthinned versions. The third possibility is to combine

[3] These are *spec*s of the form (FUNCTORspec *funid strid sigexp sigexp*).

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:

```
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.

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 our new definition of thinning an environment by an interface:

$$(FE,\ MSE,\ VE,\ EE) \downarrow (FIE,\ SIE,\ vars,\ excons) = (FE',\ MSE',\ VE',\ EE')$$

where

$$\begin{aligned} FE' = \{\ & funid \mapsto (strid,\ int,\ strexp,\ int_2,\ B)\ ;\\ & FE\ (funid) = (strid,\ int,\ strexp,\ int_1,\ B) \text{ and } FIE\ (funid) = int_2\ \} \end{aligned}$$

and, as in SML,

$$MSE' = \{strid \mapsto ME{\downarrow}I\ ;\ MSE\ (strid) = ME \text{ and } SIE\ (strid) = I\,\}$$

and VE' and EE' are obtained by restricting the domains of VE and EE to *vars* and *excons*, respectively.

Evaluation Generally we obtain the evaluation rules for the new language by modifying the evaluation rules of SML to work with the new semantic objects and functions in the obvious way. We must also add new rules to deal with the syntax we have added, and make significant changes to some other rules. We give these rules in Fig. 5, along with pointers to relevant rules in the SML semantics for readers with access to a copy of the *Definition*. Here we describe the rules in Fig. 5:

1. This rule defines how to evaluate the application of a *longfunid* to a *strexp*. As in SML, evaluating a *strexp* can result in an exception packet p. These cause no complications so we do not discuss them. [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, evaluate the declaration in this environment, and then lift any resulting Core environment to a Module environment to get the final result. [Replaces rule 164.]
3. This give the semantics of the Module-level `open` declaration. [Follows rule 166.]
4. This rule shows how to evaluate a functor declaration, now that functor declarations are specific instances of module declarations. [Follows rule 168.]
5. This describes how to evaluate a functor specification. [Follows rule 183].
6. This rule gives the semantics of rebinding a functor to a new identifier. [Follows rule 187.]

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

$$\frac{B\ (\mathit{longfunid}) = (\mathit{strid} : I, \mathit{strexp}'\langle : I'\rangle, B') \qquad B \vdash \mathit{strexp} \Rightarrow \mathit{ME}_1 \quad B' + \{\mathit{strid} \mapsto \mathit{ME}_1{\downarrow}I\} \vdash \mathit{strexp}' \Rightarrow \mathit{ME}_2}{B \vdash \mathit{longfunid}(\mathit{strexp}) \Rightarrow \mathit{ME}_2\langle{\downarrow}I'\rangle} \tag{1}$$

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

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

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

$$\frac{B(\mathit{longstrid}_1) = \mathit{ME}_1 \ \cdots \ B(\mathit{longstrid}_n) = \mathit{ME}_n}{B \vdash \texttt{open}\ \mathit{longstrid}_1 \ \cdots \ \mathit{longstrid}_n \Rightarrow \mathit{ME}_1 + \cdots + \mathit{ME}_n} \tag{3}$$

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

$$\frac{B \vdash \mathit{funbind} \Rightarrow F}{B \vdash \texttt{functor}\ \mathit{funbind} \Rightarrow F \text{ in Modenv}} \tag{4}$$

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

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

$\boxed{B \vdash \mathit{funbind} \Rightarrow F}$

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

Fig. 5. Evaluation rules for higher-order functors

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. For other phrase classes and semantic objects we distinguish the names of the HOF-ML constructors and types from those of Mod-ML 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. 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*). At present, we have not finished proving this fact. Before we discuss how we prove such a result, we need to discuss what result we are trying to prove. Most concisely, we aim to prove that there is a function `embed_topdec` mapping top-level declarations of Mod-ML into top-level declarations of HOF-ML, a function `embed_basis` mapping bases from Mod-ML into bases of HOF-ML, and a function `proj_basis_h` mapping bases (or exception packets) of HOF-ML back to bases (or exception packets) from Mod-ML such that

- For each basis $\mathtt{B}_1$, state $\mathtt{s}_1$ and top-level declaration `top_dec` of Mod-ML, if there exists a state $\mathtt{s}_2$ and a basis (or packet) $\mathtt{B}_2$ of Mod-ML such that

  $$\mathtt{eval\_topdec\ top\_dec\ s_1\ B_1\ s_2\ B_2}$$

  holds, then there exists a state $\mathtt{s}'_2$ and a basis (or packet) $\mathtt{B}'_2$ of HOF-ML such that

  $$\mathtt{eval\_topdec\_h\ (embed\_topdec\ top\_dec)\ s_1\ (embed\_basis\ B_1)\ s'_2\ B'_2}$$

  also holds, and
- For each basis $\mathtt{B}_1$, states $\mathtt{s}_1$ and $\mathtt{s}_2$, top-level declaration `top_dec` of Mod-ML, and basis (or packet) $\mathtt{B}_2$ of HOF-ML, if

  $$\mathtt{eval\_topdec\_h\ (embed\_topdec\ top\_dec)\ s_1\ (embed\_basis\ B_1)\ s_2\ B_2}$$

  holds, then

  $$\mathtt{eval\_topdec\ top\_dec\ s_1\ B_1\ s_2\ (proj\_basis\_h\ B_2)}$$

  also holds.

Informally, this states that if you wish to evaluate a top-level 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-level declarations and bases. However just to define the functions `embed_topdec`, `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 flavor of the embedding by showing three of the mutually recursive clauses for 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 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.

Projecting HOF-ML syntax back into Mod-ML is somewhat more complicated. This is because we have merged two classes, `strdec` and `fundec`, in Mod-ML into one class, `moddec` in HOF-ML, and now we are going to have to tease them apart again. In fact, this problem prevents the collection of functions briefly described above from actually being true embeddings: both `EMPTYstrdec` and

`EMPTYfundec` get mapped to `EMPTYmoddec`. In attempting to "project" HOF-ML back to Mod-ML, we cannot necessarily determine which `EMPTYmoddec`s came from `EMPTYstrdec` and which from `EMPTYfundec`. Sequences of Module declarations are another source of ambiguity. What do we do with a sequence of Module declarations that contains both structure declarations and functor declarations? In an arbitrary manner we choose to map the sequence to a sequence of the same kind as the leftmost declaration in the sequence, mapping declarations of a different kind to an empty declaration. While this complicates the definition of the projection function for declarations, and causes a loss of information, there is no harm in it since no such mixed sequence could be the result of an embedded sequence from Mod-ML.

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 primitive mutually recursive specifications over those types.

Proving Conservativity

Although the result relating the evaluation of top-level declarations in the two Module systems mentioned above is our main statement of conservative extension, in order to prove such a result we need to prove corresponding results for all layers of the evaluation relations. Therefore, to simplify the process we begin with showing the corresponding results for signature expressions, descriptions, and specifications, and work our way up through the syntax classes. To further simplify the process, we show each of the two parts of the main conservativity theorem separately. The next step in proving the results is to coerce each of the implications into a form that we can use with our induction principles. For example, the second clause for `topdec` becomes:

$$\begin{array}{l}\forall\, \texttt{topdec\_h}\ s_1\ B\ s_2\ \texttt{bp}.\ \texttt{eval\_topdec\_h topdec\_h}\ s_1\ B\ s_2\ \texttt{bp} \Rightarrow \\ \ \forall\, \texttt{topdec}\ B'.\ ((\texttt{topdec\_h} = \texttt{embed\_topdec topdec}) \wedge (B = \texttt{embed\_basis}\ B')) \Rightarrow \\ \quad \texttt{eval\_topdec topdec}\ s_1\ B'\ s_2\ (\texttt{proj\_basis\_pack bp})\end{array}$$

Once we perform this transformation, we can apply our induction principle to reduce our problem to showing that the conclusion of the resulting implication holds for all the evaluation rules. To show these results, we have various tools at our disposal, including structural induction over both the syntax and the semantics, rewriting with theorems that state the distinctness of all the constructors, and rewriting with the equations stating the mutually 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 these results also at our disposal when proving the later results.

While there is a great deal of regularity involved in carrying out these proofs, it is not apparent at present that we could write a general purpose tactic that would automatically prove all of these theorems. Each case seems to have just enough that is distinct about it to benefit from interactive guidance.

5 Conclusion

We have outlined how we used the interactive theorem prover HOL to give the dynamic specification 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 done 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 as fast, maybe faster, with the theorem prover, as we could formulate 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 so did the theorem prover receive benefits from us. The specification task has motivated us to improve HOL's handling of mutually recursive types, and to write a general purpose tactic suitable for defining mutually recursive families of relations and deriving the appropriate induction principles.

Both the benefits to the specification task and to the theorem prover were made possible by a combination in HOL of an expressive language capable of developing much general mathematics, with an open yet secure system which allows users to develop theorem-proving methodologies to suit their needs.

References

1. David B. MacQueen, Mads Tofte. *A Semantics for Higher-order Functors*, unpublished draft. Contact Dave MacQueen at AT&T Bell Labs, Room 2a-431, 600 Mountain Ave, Murray Hill, NJ 07974, USA.
2. Thomas F. Melham. *A Package for Inductive Relation Definitions in HOL*, Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications. IEEE Computer Society Press, 1992.
3. Robin Milner, Mads Tofte. *Commentary on Standard ML*, The MIT Press, Cambridge, Mass, 1991.
4. Robin Milner, Mads Tofte, Robert Harper. *The Definition of Standard ML*, The MIT Press, Cambridge, Mass, 1990.
5. Myra VanInwegen, Elsa Gunter, HOL-ML. Proceedings of HUG '93. Lecture Notes in Computer Science 780, Springer-Verlag, 1994.

This article was processed using the LaTeX macro package with LLNCS style