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. Building on work done by Myra VanInwegen and Elsa Gunter, we encode the dynamic semantics of the Module system of SML in HOL. We extend this to a possible dynamic semantics for a Module system with higher order functors. We relate these two semantics by embeddings and projections and outline how we are using these to state and prove that the new evaluation relations conservatively extend the old ones.

1 Introduction

This paper describes the latest stage in an ongoing program of using the HOL theorem-prover to study the semantics of the ML programming language. We encode the specification of the dynamic semantics of the Module system for Standard ML, and a speculative specification of the dynamic semantics of a Module system with higher-order functors for an extension of Standard ML. To the best of our knowledge the dynamic semantics for the Module system with higher-order functors has not been written down before. We chose to give these semantics in the setting of a theorem prover in order to use the theorem prover to assist us in proving theorems that would provide evidence that our proposed semantics are sensible. Such theorems include that the evaluation relations in the dynamic semantics of the extended Module system conservatively extend those in the dynamic semantics of the original system.

As was done in encoding of the dynamic semantics of the core language, we encode these semantics via definitions and properties derived from these definitions, as opposed to taking an axiomatic approach. The definitional approach requires a higher-order language. However it provides us with induction principles which allow us to prove facts about evaluation which do not follow from the rules, including negative facts such as that certain expressions do not evaluate. Moreover, these induction principles are critical tools in proving "meta"-facts about the evaluation relations, such as the conservative extension result.

* Partially supported by AT&T

2 Mod-ML

In this paper we often make references to section numbers in *The Definition of Standard ML* [4] which we mark by enclosing them in square brackets. We will refer to [4] itself as the *Definition*.

Mod-ML is an extension of the system HOL-ML, described in [5]. HOL-ML is an encoding of the dynamic semantics of a major subset of the Core language of SML in the theorem-prover HOL. SML syntax and the semantic objects used in the *Definition* 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. All of the definitions that make up HOL-ML must be loaded into HOL in order for Mod-ML to be loaded.

The syntactic terms of Mod-ML consist of those of HOL-ML augmented with the terms generated by the grammar for the Module system (Fig. 1.) This grammar produces a reduced syntax for the ML Module system, appropriate to the requirements of the dynamic semantics. This syntax is derived from that presented in [3.4] according to the procedure described in [7.1]. Each of the phrase classes in this grammar is represented as 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, according to the dependencies that exist, and define these separately. We believe that by avoiding unnecessary mutual recursion among the types the presentation of definitions and theorems will be clearer and inductive proofs will be simpler. To be able to make these definitions in the form in which they appear in (Fig. 1, we have made use of 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 the previous package for mutual recursion developed by Elsa Gunter and Myra VanInwegen and originally used in the development of HOL-ML.

The semantic objects of Mod-ML are those of HOL-ML plus those necessary for the evaluation of signatures, structures, and functors. These additional semantic objects are presented in Fig. 2. These are intended to represent the semantic objects described in [7.2]. The object classes `intenv`, `sigenv`, and `funenv` call for finite function spaces which, following [5], we represent as lists of pairs of the appropriate identifier class and value class.

Next we define several functions on semantic objects, to represent the operations decribed in [4.3] and [7.2]. These include projecting components of tuples; injecting components into tuples; modification of environments, bases and the like; extracting interfaces from environments and interface bases from bases; cutting down environments according to interfaces. In the case of those functions which are intended to operate on finite function spaces we are always careful to ensure that the lists are maintained in lexicographical ordering by idenitfiers. Our aim here is to make the list structure transparent, so that we adequately

```
 (* Basic datatype constructors *)
'a option        ::= NONE | SOME 'a
'a nonemptylist  ::= ONE 'a | MORE 'a ('a nonemptylist)
'a long          ::= BASE 'a | QUALIFIED strid ('a long)

 (* Identifiers *)
sigid   ::= SIGID string
funid   ::= FUNID string

 (* Descriptions and Specifications *)
valdesc ::= VARvaldesc var (valdesc option)

exdesc  ::= EXCONexdesc excon (exdesc option)

sigexp  ::= SIGsigexp spec | SIGIDsigexp sigid
spec    ::= VALspec valdesc | EXCEPTIONspec exdesc |
            STRUCTUREspec strdesc | LOCALspec spec spec |
            OPENspec ((strid long) nonemptylist) |
            INCLUDEspec (sigid nonemptylist) |
            EMPTYspec | SEQspec spec spec
strdesc ::= STRIDstrdesc strid sigexp (strdesc option)

sigbind ::= BINDsigbind sigid sigexp (sigbind option)

sigdec  ::= SIGNATUREsigdec sigbind | EMPTYsigdec |
            SEQsigdec sigdec sigdec

 (* Structure and Signature Expressions *)
strexp  ::= STRUCTstrexp strdec | LONGSTRIDstrexp (strid long) |
            APPstrexp funid strexp | LETstrexp strdec strexp
strdec  ::= DECstrdec dec | STRUCTUREstrdec strbind |
            LOCALstrdec strdec strdec | EMPTYstrdec |
            SEQstrdec strdec strdec
strbind ::= BINDstrbind strid (sigexp option) strexp (strbind option)

 (* Functors and Top-Level Declarations *)
funbind ::= BINDfunbind funid strid sigexp
                        (sigexp option) strexp (funbind option)
fundec ::= FUNCTORfundec funbind | EMPTYfundec | SEQfundec fundec fundec
topdec ::= STRDEC strdec | SIGDEC sigdec | FUNDEC fundec
```

Fig. 1. Mod-ML grammar

represent finite functions.

To complete the encoding we define the evaluation relations which define how the terms of each phrase class evaluate to the appropriate semantic object. Again we follow 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: first we define a predicate on possible evaluation relations for the phrase classes in the group which says if the possible evaluation relations satisfy the rules of evalution the the phrase classes; then we define each evaluation relation as the smallest relation of the appropriate type satisfying the evaluation-relation predicate. We have one such evaluation-relation predicate for each group of mutually inductive types. By "the smallest relation", we mean that the evaluation relations are defined as the logical intersection of all relations satisfying that predicate. For example, the relation `eval_strexp` is defined as:

```
eval_strexp strexp s1 B s2 ep =
  ∀ 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 s1 B s2 ep
```

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 composistion of built-in tactics parameterized by the definitions.

```
int      ::= INT intenv (var set) (excon set)
intenv   ::= INTENV ((strid # int) list)

sigenv   ::= SIGENV ((sigid # int) list)

intbasis ::= INTBASIS sigenv intenv

funclos  ::= FUNCLOS strid int strexp (int option) basis
funenv   ::= FUNENV ((funid # funclos) list)
basis    ::= BASIS funenv sigenv env
```

Fig. 2. Mod-ML evaluation semantic objects

3 HOF-ML

It has been proposed (Section 8.5 of [3], [1]) to extend Standard ML by allowing functors to take functors as arguments, and to be included 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 attempt to work out what the dynamic semantics of this extension should be, and then use HOL to explore the relationship between the extended system and the original system.

3.1 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. To obtain the Module language grammar, start with Fig. 1. Replace the phrase classes `fundec` and `strdec` with the class `moddec` from Fig. 3. Add to the grammars of `spec` and `funbind` the clauses shown in Fig. 3. Replace the `APPstrexp` clause of the grammar of `strexp` by the version in Fig. 3. Replace the grammar of `topdec` by the version in Fig. 3. These changes are explained here:

Structure expressions (`strexp`) We can now apply functors defined within the bodies of structures.

Module declarations (`moddec`) We amalgamate structure declarations and functor declarations into one large class of Module declarations. We also add a version of `open` that exposes functor bindings to the top-level, and thus must be a Module declaration. Note that there still remains a version of `open` in the Core language which does not expose functor bindings.

Specifications (`spec`) It is now possible to specify a functor. This is done by giving its name, the name of its input structure, and signatures specifying its input and output structures.

Functor bindings (`funbind`) It appears to be an omission in the SML grammar that no provision is made for rebinding a functor to another functor identifier. We remedy this here, since this language feature is important to us as we can use it to give top-level names to functors occurring within structures and to rebind functors passed through a functor's parameter.

Encoding the syntax We encode the new syntax into HOL as a number of inductive type definitions, just as we did in Mod-ML. The new versions of the phrase classes `sigid`, `funid`, `valdesc`, and `exdesc` are identical to the old ones, so we simply re-use the types used to encode these in Mod-ML. In the encoding we distinguished the names of the constructors and types for HOF-ML from those

```
spec    ::= ... | FUNCTORspec funid strid sigexp sigexp
strexp  ::= ... | APPstrexp (funid long) strexp | ...
moddec  ::= DECmoddec dec | STRUCTUREmoddec strbind |
            LOCALmoddec moddec moddec |
            OPENmoddec ((strid long) nonemptylist) |
            EMPTYmoddec | SEQmoddec moddec moddec |
            FUNCTORmoddec funbind
funbind ::= ... | REBINDfunbind funid (funid long)
topdec  ::= MODDEC moddec | SIGDEC sigdec
```

Fig. 3. HOF-ML grammar (additions and changes)

of Mod-ML by appending `_h` to them. Thus, for example, the type representing HOF-ML structure decriptions 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.

3.2 Semantic objects

Whereas deciding on a new syntax was relatively straightforward, choosing appropriate semantic objects is a messier task. Deciding what environments should be is the main difficulty. In Mod-ML, environments play two roles in the dynamic semantics. They tell us what values are associated with given long identifiers during the evaluation of a Core expression, and they are the values returned by the evaluation of structure expressions. Since structures contain only structure declarations and Core language declarations in Mod-ML, 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: we can either use environments in the Core dynamic semantics which have an excess of information, or we can have 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. Therefore, we decided the second option was most in keeping with the nature of the specification given in the *Definition*, and this is the choice we have pursued here. In future work, we intend to encode the alternate option and prove that the two approaches 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 for the Core language and one for the Module system. When an `open` declaration occurs within a collection of local bindings in a Core language expression, it must create a Core language environment; it must cut out all functor information fron the structures it is opening. However, when an `open` declaration occurs in the body of a structure it should keep all functor information in the structures being opened, since those functors may be used to created new substructures. Thus we have added a version of `open` to the Module declarations, in addition to leaving the one in the Core declarations. 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.

The semantic objects of HOF-ML are specified by the grammar in Fig. 4, plus the classes `sigenv`, `funclos` and `funenv` which remain unchanged from Fig. 2. As was the case with the syntax, in the encoding of the semantic objects we distinguished the names of the constructors and types for HOF-ML from those of Mod-ML by tagging them with `_h`. Here we describe the additions and changes:

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 generally 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`).

Interfaces (`int`) Interfaces are prescriptions for how to cut down the view of a structure. Since structures may now contain functors, interfaces must now prescribe how to cut down the view of a functor. Therefore they contain a new component — a functor interface environment.

Structure Interface Environments (`strintenv`) These are the semantic objects which were called interface environments in SML. We have renamed them to more accurately reflect their function in HOF-ML.

Functor Interface Environments (`funintenv`) Our choice of interface information for functors is discussed at length in Section 3.3.

Bases (`basis`) Bases no longer need to contain a separate functor environment component, since this has been moved into the `modenv` component.

Interface Bases (`intbasis`) These now have a new component — a functor interface environment.

3.3 Functions on semantic objects

Most of the projection, injection, and modification functions on HOF-ML semantic objects can be defined by straightforward modifications to the corresponding Mod-ML functions. Here we describe those functions that are significantly different. For readability, we present definitions in the notation used in the *Definition*,

```
int       ::= INT funintenv strintenv (var set) (excon set)
strintenv ::= STRINTENV ((strid # int) list)
funintenv ::= FUNINTENV ((funid # int) list)
intbasis  ::= INTBASIS funintenv sigenv strintenv
modenv    ::= MODENV funenv modstrenv varenv exconenv
modstrenv ::= MODSTRENV ((strid # modenv) list)
basis     ::= BASIS sigenv modenv
```

Fig. 4. HOF-ML evaluation semantic objects (additions and changes)

but the functions described all exist as terms in HOL. We use the variable naming conventions of the *Definition*: *FIE* and *SIE* range over functor and structure interface environments respectively; *ME*, *FE*, *G*, *VE* and *EE* range over Module-level, functor, signature, variable, and exception environments respectively, and *MSE* ranges over Module-level structure environments.

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:

$$\mathsf{Inter}\ (\mathit{FE},\ \mathit{MSE},\ \mathit{VE},\ \mathit{EE}) = (\mathit{FIE},\ \mathit{SIE},\ \mathsf{Dom}\ \mathit{VE},\ \mathsf{Dom}\ \mathit{EE})$$

where

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

and, as before,

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

That is, for functor and module structure environments, the interface we extract is the set of mappings from the identifiers they contain to interfaces for the values they associate with the identifiers. For variable and exception environments, the corresponding interfaces are just the sets of identifiers they contain. As before, we extend Inter to extract an interface basis from a basis. We diverge from the *Definition* in giving this function its own name, Inter_basis. It is defined as follows:

$$\text{Inter\_basis}\ (\mathit{G},\ \mathit{ME}) = (\mathit{FIE}\ \text{of}\ (\text{Inter}\ \mathit{ME}),\ \mathit{G},\ \mathit{SIE}\ \text{of}\ (\text{Inter}\ \mathit{ME}))$$

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 every 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.

It is worth reflecting 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_h`). 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.

Cutting down environments Interfaces become more complicated in the setting of higher-order Modules because they must contain information concerning how to cut down the "view" of a functor. In 3.2 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 cut down (or *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

[3] This is a `spec` of the form (`FUNCTORspec funid strid sigexp sigexp`).

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 both the first and the second definitions of thinning.

We belive 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 would find that `A1.z` = `5` and `A2.z` = `6`, whereas it should be the case that `A1.z` = `5` = `A2.z`. For the computation of `z`, F requires the `x` in the top-level environment be used, and this should remain the case, even if we subsequently thin F. Thinning should only change 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.

Now we define the operation $\downarrow$ of cutting down an environment to an interface:

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

where

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

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.

3.4 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 the

functions on them) in a fairly obvious manner. We must also add some new rules to deal with the syntax which we have added, and make major changes to some other rules. In figure 5 we give the rules which are new or have been changed significantly. Here we describe these rules:

1. This rule takes the place of rule 162. The change here reflects the fact that we now have long functor identifiers.
2. This rule replaces rule 164. 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 obtain the final result.
3. This rule goes just after rule 166. It is a version of rule 132 appropriate to Module-level declarations. It shows how to evaluate a Module-level `open` declaration.
4. This rule goes just after rule 168. It is a version of rule 188 modified to return a module environment instead of a functor evironment. It shows how to evaluate a functor declaration, given that functor declararions are now specific instances of Module declarations.
5. This rule describes how to evaluate a functor specification. It should be inserted just after rule 183.
6. This rule goes after rule 187. It gives the semantics of rebinding a functor to a new identifier.

4 Relating Mod-ML and HOF-ML

So far we have described the encoding of two possible Module systems to extend the Core language of Standard ML. 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

$$\mathtt{eval\_topdec\ top\_dec\ s}_1\ \mathtt{B}_1\ \mathtt{s}_2\ \mathtt{B}_2$$

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

$$\mathtt{eval\_topdec\_h\ (embed\_topdec\ top\_dec)\ s}_1\ \mathtt{(embed\_basis\ B}_1\mathtt{)}\ \mathtt{s}'_2\ \mathtt{B}'_2$$

also holds, and

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. HOF-ML new evaluation rules

– 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

  `eval_topdec_h (embed_topdec top_dec)` $\mathtt{s}_1$ `(embed_basis` $\mathtt{B}_1$`)` $\mathtt{s}_2$ $\mathtt{B}_2$

  holds, then

  `eval_topdec top_dec` $\mathtt{s}_1$ $\mathtt{B}_1$ $\mathtt{s}_2$ `(proj_basis_h` $\mathtt{B}_2$`)`

  also holds.

Informally, this states that if you wish to evaluate a top-level declaration of Mod-ML, it suffices 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.

4.1 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 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.

4.2 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 comes from the fact that we 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 left most declaration in the sequence, mapping declarations of a different kind to an empty declaration. While it complicates

the definition of the projection function for declarations, and causes a loss of information, there is no harm in this 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.

4.3 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:

$$\begin{array}{l}\forall\, \texttt{topdec\_h}\ \texttt{s}_1\ \texttt{B}\ \texttt{s}_2\ \texttt{bp}.\ \texttt{eval\_topdec\_h topdec\_h}\ \texttt{s}_1\ \texttt{B}\ \texttt{s}_2\ \texttt{bp} \Rightarrow \\ \quad \forall\, \texttt{topdec}\ \texttt{B}'.\ ((\texttt{topdec\_h} = \texttt{embed\_topdec topdec}) \wedge (\texttt{B} = \texttt{embed\_basis}\ \texttt{B}')) \Rightarrow \\ \qquad \texttt{eval\_topdec topdec}\ \texttt{s}_1\ \texttt{B}'\ \texttt{s}_2\ (\texttt{proj\_basis\_pack bp})\end{array}$$

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 Standard ML, 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.
2. Thomas F. Melham. *A Package for Inductive Relation Definitions in HOL*, HOL library.
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. Presented at HUG '93.

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