HOL-ML

Myra VanInwegen*[1] and Elsa Gunter[2]

[1] University of Pennsylvania Computer and Info Science Dept
Philadelphia PA 19104, USA
myra@saul.cis.upenn.edu
[2] AT&T Bell Laboratories, Rm.#2A-432
600 Mountain Ave., Murray Hill, N.J. 07974, USA
elsa@research.att.com

Abstract. We describe here HOL-ML, an encoding of a subset of SML and its dynamic semantics (as described by *The Definition of Standard ML*) in HOL. This encoding, which is the first stage in a project that will include typechecking and SML Modules, allows the formal study of the evaluation of a real programming language including state and control constructs (exceptions). In this paper we describe the subset of SML that we encoded and the semantic objects needed for its evaluation. We explain how we defined the evaluation rules and how we proved that evaluation is deterministic. We describe briefly the next step, which is to define a larger language that includes type declarations and to define the typechecking rules on it. Finally, we give a short description of the mutually recursive type definition package that we wrote to enable us to define the types we needed to create the HOL-ML grammar.

1 Introduction

This paper describes HOL-ML, our encoding of the dynamic Core of SML in the HOL interactive theorem prover. The purpose of this work is to act as a foundation for a system for formally specifying and reasoning about SML programs. The work described here deals only with the dynamic Core language and the evaluation of programs expressed in it. As such it is one piece of a larger project that will eventually include a system for reasoning about elaboration (*e.g. type-checking*) and evaluation of programs from both the Core and the module system. Even though it is intended that this work be a piece of a larger whole, it is already a system allowing us to carry out significant reasoning about SML. This work enables us to rigorously prove routinely assumed facts, such as that evaluation as described in [3] is deterministic. It also gives us a framework for rigorously proving that two expressions evaluate to the same value. Such an ability would be necessary, for example, to use the system for safely optimizing HOL described in [4]. Perhaps most significantly, this work will allow us to state and prove facts about programs that alter state, including local state. While we have only begun to carry out exercises such as those mentioned above, the

* Partially supported by AT&T

encoding of the formal semantics of the dynamic Core of SML and its evaluation relations is a necessary major first step.

In the remainder of this section, we describe the subset of SML we encoded. In Sect. 2 we describe in detail the HOL-ML grammar and semantic objects (the states, environments, and possible values for HOL-ML phrases). In Sect. 3 we describe how we defined the evaluation rules. In Sect. 4 we explain how we proved that evaluation is deterministic. In Sect. 5 we briefly describe how encoding the static Core language and defining typechecking will be handled in a manner analogous to the work described here, and how the static Core relates to the dynamic Core. In the appendix there is a brief description of the mutually recursive type definition package we created to enable us to define the types needed for the HOL-ML grammar.

We encoded a subset of Standard ML, as described in *The Definition of Standard ML* ([3]). We often refer to this book as the *Definition* and enclose references to it in brackets ([·]). Our starting point is the syntax of the Core [Chap. 2]. This work only deals with the dynamic Core of SML and the evaluation relations. As such we restrict our language to the reduced syntax described in [Sect. 6.1]. This reduction removes all type information, including declarations of new types, type tags in expressions and patterns, and indications of argument types for exception constructors. The dynamic Core also lacks information concerning fixity of operators: it is assumed that infixes have been resolved during the parsing phase. Thus we also eliminate rules pertaining to the declaration and use of infixes. One further reduction we made, which is not in keeping with the *Definition*, is that we pared down the variety of basic datatypes, allowing only integers and strings (not reals).

We do not enforce the syntactic restrictions of [Sect. 2.9], such as the restriction that a pattern may not contain the same variable more than once; however we assume in our definition of the evaluation rules that program phrases satisfy these restrictions. Similarly, we assume that the programs are type-correct, in the sense that they are the reduced-syntax versions of type-correct programs. In Sect. 5 of this paper we describe more clearly the relationship between the full syntax and the reduced one we use here when we examine the typechecking application.

The syntax of the Core is primitive, not including derived syntax such as `fun` declarations, boolean operators, and the `if` construct. As HOL-ML language phrases are created with constructors rather than parsed from some concise concrete syntax as is the case with SML compilers, HOL-ML programs are large and quite complicated to read. However, it is possible to write a parser and pretty-printer for HOL-ML so that programs could be entered using SML syntax.

In addition to encoding a description of the dynamic Core grammar, we also need an encoding of the semantic objects involved in the evaluation of the dynamic Core. These are discussed in the next section.

2 HOL-ML Grammar and Semantic Objects

The HOL-ML grammar and semantic object definitions are presented in Backus-Naur Form (BNF) in Figs. 1 and 2. Each of the language phrases and semantic objects represents an HOL type (the appendix explains how to transform these descriptions into HOL type definitions).

```
var   ::= VAR string
con   ::= CON string
scon  ::= SCINT integer | SCSTR string
excon ::= EXCON string
label ::= LABEL string
strid ::= STRID string

'a nonemptylist  ::= ONE 'a | MORE 'a ('a  nonemptylist)
'a long          ::= BASE 'a | QUALIFIED strid ('a long)

atexp   ::= SCONatexp scon | VARatexp (var long) | CONatexp (con long) |
            EXCONatexp (excon long) | LETatexp dec exp | PARatexp exp |
            RECORD1atexp | RECORD2atexp exprow
exprow  ::= EXPROW1 label exp | EXPROW2 label exp exprow
exp     ::= ATEXPexp atexp | APPexp exp atexp | HANDLEexp exp match |
            RAISEexp exp | FNexp match
match   ::= MATCH1 mrule | MATCH2 mrule match
mrule   ::= MRULE pat exp
dec     ::= VALdec valbind | EXCEPTdec exbind | LOCALdec dec dec |
            OPENdec ((strid long) nonemptylist) | SEQdec dec dec | EMPTYdec
valbind ::= PLAIN1valbind pat exp | PLAIN2valbind pat exp valbind |
            RECvalbind valbind
exbind  ::= EXBIND1 excon | EXBIND2 excon exbind |
            EXBIND3 excon (excon long) | EXBIND4 excon (excon long) exbind
atpat   ::= WILDCARDatpat | SCONatpat scon | VARatpat var |
            CONatpat (con long) | EXCONatpat (excon long) | RECORD1atpat |
            RECORD2atpat patrow | PARatpat pat
patrow  ::= DOTDOTDOT | PATROW1 label pat | PATROW2 label pat patrow
pat     ::= ATPATpat atpat | CONpat (con long) atpat |
            EXCONpat (excon long) atpat | LAYEREDpat var pat
```

Fig. 1. HOL-ML grammar

The grammar in Fig. 1 refers to the HOL types `integer` and `string`, both of which are defined in libraries that must be loaded prior to defining the grammar.

Definitions for the identifiers `var` (variables), `con` (value constructors), `scon` (special constants), `excon` (exception constructors), `label` (labels) and `strid` (structure identifiers) are presented before the actual grammar. The polymorphic types of `nonemptylist` and `long` are also given before the grammar. The type of `nonemptylist` is used for those constructs in the grammar that take a collection of one or more arguments of predefined type. The type `long` is used to construct various kinds of long identifiers. The HOL-ML grammar itself consists of the language phrases `atexp` (atomic expressions), `exprow` (expression rows, which form record expressions), `exp` (expressions), `match` (lists of alternatives that form function definitions), `mrule` (match rules), `dec` (declarations), `valbind` (value bindings), `exbind` (exception bindings), `atpat` (atomic patterns), `patrow` (pattern rows, which form patterns that match records), and `pat` (patterns).

Below we sometimes give the SML concrete syntax version of a language construct. This is for sake of exposition only; HOL-ML uses only the abstract syntax in Fig. 1.

There are more well-formed terms in HOL-ML than are possible results of parsing and elaboration of concrete SML syntax. An example is that the parsing rules of the *Definition* [Sect. 2.4] prevent a string from being used as a variable when it is in the scope of a constructor using the same string. In the declaration `val C = (5,C)`, parsing prevents the first `C` from being a variable while the second is a constructor. In HOL-ML this is not a problem, since the constructors (`VAR` and `CON`) distinguish them. In addition, it is obvious that we can construct terms in this grammar that cannot come from typechecked terms. As an eventual consequence, most theorems we will want to prove in later uses of HOL-ML will need to carry with them the hypothesis that the appropriate HOL-ML expressions are the translation of type-checked expressions from the full Core.

The definition of the semantic objects pose a few problems not present for the grammar. The semantic objects form a mutual recursion, as does the grammar. However, the structure of this recursion is more complex than that of the grammar. Several of the semantic types are given as finite functions between others which are mutually recursive with those types. For example, in [Fig. 13], a record is described as a finite map from labels to values, while a value is described as a disjoint sum of records with other things. This kind of mutual recursion is well beyond the scope of any package for defining mutually recursive types that currently exists, and it seemed that trying to support this would take us too far afield. Therefore, wherever the *Definition* specifies the type of semantic objects as finite functions, we have used lists of pairs instead. In each case, it is possible to give a linear ordering on the elements in the domain of the desired finite functions, and we have used this ordering to define functions for inserting pairs into the lists so that only canonical lists will be generated during evaluation. If, at a later time, support is added to HOL for such complex mutual recursions as ones involving recursive occurrences within finite function type constructors, then our encoding can easily be re-encoded with finite functions replacing lists and function update replacing insertion.

The other problem with defining the types of semantic objects arises from the fact that several of them are left under-specified in the definition. Both addresses and exception names are only specified as infinite types. We could parameterize

```
sval          ::= SVINT integer | SVSTR string
addr          ::= ADDR num
basval        ::= Size | Chr | Ord | Explode | Implode |
                  Abs | Div | Mod | Neg | Times | Plus | Minus |
                  Eql | Noteql | Less | Greater | Lesseql | Greatereql
exname        ::= EXNAME num
val           ::= ASSGval | SVALval sval | BASval basval | CONval con |
                  APPCONval con val | EXVALval exval | RECORDval record |
                  ADDRval addr | CLOSUREval closure
record        ::= NONErec | SOMErec label val record
exval         ::= NAMEexval exname | NAMEVALexval exname val
pack          ::= PACK exval
closure       ::= CLOSURE match env varenv
mem           ::= NONEmem | SOMEmem addr val mem
exnameset     ::= EXNAMESET (exname set)
state         ::= STATE mem exnameset
env           ::= ENV strenv varenv exconenv
strenv        ::= NONEstrenv | SOMEstrenv strid env strenv
varenv        ::= NONEvarenv | SOMEvarenv var val varenv
exconenv      ::= NONEexconenv | SOMEexconenv excon exname exconenv
val_pack      ::= VALvp val | PACKvp pack
record_pack   ::= RECORDrp record | PACKrp pack
val_pack_fail ::= VALvpf val | PACKvpf pack | FAILvpf
env_pack      ::= ENVep env | PACKep pack
varenv_pack   ::= VARENVvep varenv | PACKvep pack
exconenv_pack ::= EXCONENVeep exconenv | PACKeep pack
varenv_fail   ::= VARENVvef varenv | FAILvef
```

Fig. 2. HOL-ML evaluation semantic objects

by these, but we chose to use a concrete instance instead; we chose to represent represent addresses (`addr`) and exception names (`exname`) as natural numbers wrapped with appropriate constructors.

We conclude this section with a quick summary of the rest of the semantic objects. A special value (`sval`) is an integer or a string. Basic values (`basval`) are the basic functions bound to predefined identifiers. Values (`val`) are evaluations of expressions. Records (`record`) are evaluations of expression rows. Packets (`pack`) are raised exceptions. Closures (`closure`) are evaluations of functions. Memory (`mem`) is a finite mapping from addresses to values, represented as a list. A state (`state`) consists of memory and the exception names used. An environment (`env`) is the evaluation of a declaration. A structure environment (`strenv`) is a finite mapping from structure identifiers to environments. A variable environment (`varenv`) is the evaluation of a pattern phrase type. An exception constructor environment (`exconenv`) is the evaluation of an exception binding.

The last several semantic objects are the results of evaluating language phrases. The name of the semantic object is composed of two or three other semantic objects (or `fail`, indicating that the value doesn't match the pattern) separated by underscores, such as `val_pack`. Each of these is just the disjoint sum of the components in its name.

3 Evaluation

To define evaluation, we must define the relations, one for each language phrase in the grammar, that determine the evaluation of an HOL-ML program. These relations are named `eval_`*phrase* for each language phrase *phrase*. Each relation is an HOL function taking a collection of terms to booleans. The arguments of the functions are the language phrase being evaluated and various semantic objects, including evaluation results, states, environments, and, for patterns, the value against which the pattern is being matched. The relation for a phrase type holds if the phrase evaluates to the result (and possibly new state) in the context given by the state, environment, and (for patterns) value.

In order to define our evaluation relations we take a somewhat roundabout route. These relations are mutually recursive functions, since the evaluation of a language phrase depends on the result of the evaluation of the subparts of the phrase. Unfortunately, we could not use `define_mutual_functions` (a part of our mutually recursive types definition package that is used to define mutually recursive functions over these types) to define the evaluation relation because evaluation is not primitive recursive. To see why, look at rules (117) and (118) for function evaluation in [3]: there we evaluate a subpart (the match) of the result of a previous evaluation (the closure), not a subphrase of the phrase we're evaluating.

Instead, we define a predicate (called `eval_pred`) on potential evaluation relations (which are any collection of functions of the appropriate HOL types) which holds of those collections of relations that satisfy the evaluation rules in [Chap. 6].

3.1 Defining `eval_pred`

Examining the rules for evaluation in [Sect. 6.7] of the *Definition*, it is possible to note that there are actually three separate classes of inductively defined relations given there. There are the rules for evaluating exception bindings; the rules for evaluating atomic patterns, pattern rows, and patterns; and the rules for evaluating everything else, such as expressions and declarations. The latter class depends upon the previous two, but not the other way around. Therefore, we can break up our definitions of the rules into three separate smaller definitions of mutually inductive relations. The cases are all fairly similar and they could be handled all as one large definition. However, there are several pragmatic reasons for breaking them up. If we decompose the rules into three separate classes, we will get three smaller principles of induction to be used later for proving properties about the evaluation relations. In particular, if we wish to prove a property of the evaluation of a pattern, we will not have to consider a collection of irrelevant cases involving expressions, declarations and so on. Another pragmatic reason for breaking up the rules into these classes is to shorten the computation time required to prove that the relations we are defining actually satisfy the rules.

Therefore, to define the evaluation relations, we define three separate predicates, `eval_exbind_pred`, `eval_pat_pred` and `eval_pred`, each of which takes as arguments potential evaluation relations and return `T` (true) if the relations satisfy the evaluation rules. The biggest difference in how the definitions of these predicates are derived from the rules comes from the implicit rules concerning the generation of packets (raised exceptions) rather than values as results of evaluations of program phrases. The type information given in the *Definition* for the rules concerning the evaluation of patterns indicates that it is not possible for those evaluations to result in packets. Therefore, there are no implicit rules concerning the treatment of packets in those cases. To be clearer about all this, let us examine the most complicated predicate, `eval_pred`, in greater detail.

In order to give a general flavor of what `eval_pred` looks like, we will explain how one of the rules, Rule (107), is reflected in its conjuncts. Fig. 3 shows the general form of the term defining `eval_pred` and the three conjuncts corresponding to Rule (107).

```
eval_pred
    (eval_atexp:atexp->state->env->state->val_pack->bool)
    (eval_exprow:exprow->state->env->state->record_pack->bool)
    (eval_exp:exp->state->env->state->val_pack->bool)
    (eval_match:match->state->env->val->state->val_pack_fail->bool)
    (eval_mrule:mrule->state->env->val->state->val_pack_fail->bool)
    (eval_dec:dec->state->env->state->env_pack->bool)
    (eval_valbind:valbind->state->env->state->varenv_pack->bool) =
 . . .
(* Rule 107a *)
   (!s E. eval_atexp RECORD1atexp s E s (VALvp (RECORDval NONErec))) /\
(* Rule 107b *)
    (!s1 E s2 exprow r.
     eval_exprow exprow s1 E s2 (RECORDrp r) ==>
         eval_atexp (RECORD2atexp exprow) s1 E s2
                    (VALvp (RECORDval (add_record NONErec r)))) /\
    (!s1 E s2 exprow p.
     eval_exprow exprow s1 E s2 (PACKrp p) ==>
         eval_atexp (RECORD2atexp exprow) s1 E s2 (PACKvp p)) /\
 . . .
```

Fig. 3. Part of `eval_pred`

Rule (107) is as follows:

$$\frac{\langle E \vdash \mathit{exprow} \Rightarrow r\rangle}{E \vdash \{\langle \mathit{exprow}\rangle\} \Rightarrow \{\}\langle + \ r\rangle \text{ in Val}} \tag{107}$$

This has two explicit cases, one for evaluating an empty record and one for evaluating a record with fields. To each of these two rules we are then required to apply what the *Definition* refers to as the *state convention* and the *exception convention*. Once we have done so, the preceding two rules become the following three rules:

$$\frac{}{s, E \vdash \{\} \Rightarrow \{\} \text{ in Val}, s}$$

$$\frac{s_1, E \vdash \mathit{exprow} \Rightarrow r, s_2}{s_1, E \vdash \{\mathit{exprow}\} \Rightarrow \{\} + \ r \text{ in Val}, s_2}$$

$$\frac{s_1, E \vdash \mathit{exprow} \Rightarrow p, s_2}{s_1, E \vdash \{\mathit{exprow}\} \Rightarrow p, s_2}$$

where s, s_1, and s_2 are states and p is a packet. These three rules now correspond to the three conjuncts shown in Fig. 3.

During the course of encoding the rules of evaluation into HOL, we observed some errors and some peculiarities. The two significant problems, mentioned below, were known (see [1]) at the time, but not to us. A trivial typographical error, but one that was caught by the typechecker of HOL is a missing v on the lefthand side of the turnstile in the conclusion of Rule (126). Another problem arises with Rule (116), the rule for evaluating the application of a base constant to an expression. This rule as stated only deals with the case when the application of the corresponding base value to the expression's value results in a value; it does not handle the case when the application results in a packet being raised. Such a case is not handled by the *exception convention* and needs a rule of its own. We took the liberty of correcting these problems when encoding the rules.

A somewhat stickier problem arises in the case of evaluations involving the constructor `ref`. Note that the constructor `ref` is given special treatment in the pattern rules (154), (155), and (158), as it is in our translation of those rules. This also occurs in rules (112) and (114) for evaluating expressions. Because of this, one cannot rebind `ref` to be a constructor for another type, although the static semantics (the typechecking rules) allow it. Stephen Kahrs, in [1], suggests a way to resolve this conflict between the static and dynamic (evaluation) semantics. It requires retaining datatype definitions in the reduced syntax used for evaluation and keeping information on constructors in the variable environment. Although we became aware of this problem during our endeavors, on this point we chose to encode the rules as stated, despite their deficiency. We felt that this problem merited further study before determining what the best solution is.

We would like to take the opportunity to express our gratitude for the very considerable care and rigor that the authors put into the *Definition*. It is a truly remarkable piece of work, and without their attention to detail, our task would have been impossible.

3.2 The Evaluation Relations

Note that eval_pred only specifies when the potential evaluation relations must return T, so functions satisfying eval_pred may return T even when the rules do not justify it. Because of this we must define the evaluation relations to be the smallest relations satisfying eval_pred, that is, the intersection of all relations satisfying eval_pred. Thus, in the definition of each evaluation relation, we specify that a tuple is in the relation if and only if it is in every possible evaluation relation satisfying eval_pred. For example, the term used to define eval_exp is given as follows:

```
eval_exp ex s1 e s2 vp =
!poss_eval_atexp poss_eval_exprow poss_eval_exp poss_eval_match
 poss_eval_mrule poss_eval_dec poss_eval_valbind.
eval_pred poss_eval_atexp poss_eval_exprow poss_eval_exp
 poss_eval_match poss_eval_mrule poss_eval_dec poss_eval_valbind
   ==> poss_eval_exp ex s1 e s2 vp
```

After defining the evaluation relations, we prove that the resulting relations do indeed satisfy eval_pred. That is, the evaluation relation satisfy all the rules of the *Definition*.

In addition to the evaluation relations, to actually have the whole story for evaluation, we also give the definitions of the functions defined in the initial dynamic basis (the dynamic basis provides environments needed for evaluation) described in [Appendix D]. One of these functions (not) is shown in the next section.

3.3 Writing programs with HOL-ML

The declarations of even the simplest HOL-ML programs are long and hard to read. Take, for example, the definition of not. Its definition, using the subset of the SML grammar we've encoded is:

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

This is quite concise compared to its equivalent in HOL-ML, shown in Fig. 4.

```
VALdec
(PLAIN1valbind
 (ATPATpat (VARatpat (VAR "not")))
 (FNexp (MATCH2 (MRULE (ATPATpat (CONatpat (CON "true")))
                       (ATEXPexp (CONatexp (CON "false"))))
                (MATCH1 (MRULE (ATPATpat (CONatpat (CON "false")))
                               (ATEXPexp (CONatexp (CON "true"))))))))
```

Fig. 4. HOL-ML definition of not

As part of this project we intend to write an SML program that, given an expression and an environment and state in which to evaluate it, will figure out the result of the evaluation and final state (if the evaluation halts) and prove, using the evaluation relations, that the phrase evaluates to this.

4 Accomplished Proofs

Our largest proof to date has been the proof that evaluation is deterministic. As a major lemma, we proved that in the evaluation rules, the conclusion holds if and only if the hypothesis holds. Since `eval_pred` (or `eval_pat_pred` or `eval_exbind_pred`) holds of the rules, we know that the hypothesis implies the conclusions. The other direction is necessary for proofs by induction: in order to use facts about the evaluations of subparts in the hypothesis of the rule to prove facts about the evaluation in the conclusion of the rule, we need to know that the hypothesis holds if the conclusion does.

In order to do proofs by induction, we used a series of theorems that are based on the definitions of the evaluation rules. The "induction theorem" for expressions is as follows:

```
|- !atexp_prop exprow_prop exp_prop match_prop
     mrule_prop dec_prop valbind_prop.
     eval_pred atexp_prop exprow_prop exp_prop match_prop
       mrule_prop dec_prop valbind_prop ==>
     (!ex s1 e s2 vp. eval_exp ex s1 e s2 vp ==>
                      exp_prop ex s1 e s2 vp)
```

This theorem says that if `eval_pred` holds of some properties that have the same types as the evaluation relations, and `eval_exp` holds of some tuple, then the property on expressions, `exp_prop`, also holds of this tuple. The theorem is true because `eval_exp` is the smallest relation satisfying `eval_pred`.

In order to prove that evaluation is deterministic, we needed to prove a collection of properties, one for each phrase type, that says that given the context of an environment and a state, a phrase evaluates to at most one final state and result. The property for expressions (called `exp_det`) is as follows:

```
!ex s1 e s2 vp s2' vp'.
     (eval_exp ex s1 e s2 vp /\ eval_exp ex s1 e s2' vp') ==>
        (s2 = s2') /\ (vp = vp')
```

In order to use our induction theorem, however, we had to rephrase our goal as the following equivalent statement, called `exp_det2`:

```
!ex s1 e s2 vp. eval_exp ex s1 e s2 vp ==>
                !s2' vp'. eval_exp ex s1 e s2' vp' ==>
                            ((s2 = s2') /\ (vp = vp'))
```

Comparing this goal with the induction theorem, we see that `exp_prop` must be:

```
!s2' vp'. eval_exp ex s1 e s2' vp' ==> ((s2 = s2') /\ (vp = vp'))
```

We defined similar properties for all phrase types and then proved that `eval_pred` holds of those properties (this is the hard part). After this, it only required simple manipulations to get the determinacy properties (of which `exp_det` is one) we wanted.

5 The Next Step: Typechecking

In this section we discuss a planned extension to HOL-ML. We have encoded the dynamic Core of SML and defined its evaluation rules. However, as we have noted, these relations will work correctly only on *type-correct* programs, for some meaning of type-correct. One of two things will happen when we try to evaluate programs that are not type-correct. The first is that the program can evaluate to garbage. For example, we can prove that `(op ::)5` (the application of the *cons* operator to 5) evaluates to the value `(::, 5)`. This is because the evaluation rule

$$\frac{s, E \vdash exp \Rightarrow con, s' \quad con \neq \texttt{ref} \quad s', E \vdash atexp \Rightarrow v, s''}{s, E \vdash exp\, atexp \Rightarrow (con, v), s''} \qquad (112)$$

is applicable despite the fact that the constructor `::` ought to be applied only to records $\{$`1 =` x`, 2 =` $L\}$ where L is a list and x is an item of the same type as the items in the list.

The second thing that can happen is that the program may fail to evaluate at all because no rule applies to it. This is the case with the program

$$@\{\texttt{1} = (\texttt{op}\ ::)\texttt{5}, \texttt{2} = \texttt{nil}\}$$

which is an attempt to append `(op ::)5` to the empty list. Recall that the definition of append, without using any derived forms, is

```
val rec @ = fn {1 = nil, 2 = M} => M
             | {1 = :: {1 = x, 2 = L}, 2 = M} =>
                    ::{1 = x, 2 = @ {1 = L, 2 = M}}
```

According to the rules for evaluating function application, the first match rule is tried first, so an attempt is made to match the value to which `(op ::)5` evaluates, which is `(::, 5)`, against the pattern `nil`. This results in a FAIL, so the second match rule is tried. Here, an attempt is made to match `(::, 5)` against the pattern $\{$`1 = x, 2 = L`$\}$. The rule for evaluating record patterns has as a side condition the requirement that the value is a record. This is not the case here. Thus no rule applies, so the program does not evaluate. One of the most important theorems one can prove about typed languages is that if the program typechecks, then its evaluation will not "get stuck"; that is, in the search for a proof that the program evaluates to a value, there will always be a rule that applies. In the *Commentary on Standard ML*, the authors state that they believe this result to be true for Standard ML, but have not proved it. We hope to eventually prove it for HOL-ML. Doing so will require additional machinery such as the definition of the length of an evalution, which we do not currently have defined.

Despite the need for programs to be in some way type-correct in order to evaluate correctly, one cannot define what it means to be type-correct using the grammar of HOL-ML because it includes no type information. Thus type-correctness must be defined in terms of an expanded language that includes type declarations and type tags on language phrases. We could define the typechecking rules for this language in a manner similar to that for the evaluation rules for HOL-ML. Given this, we could define a type-correct program in this language to be one that can be proven to have a type using the typechecking rules. A translation to HOL-ML could be defined that keeps only the information relevant to evaluation. A type-correct HOL-ML program would then be defined to be the translation of a type-correct program in the expanded language into the reduced syntax of HOL-ML. It would be possible to write a function that would automatically prove to what term in HOL-ML a term in the full typed language translates, thereby easing the burden of the users of this system.

A Mutually Recursive Types

The easiest way to explain how to use our mutually recursive types definition package is to show an example. The goal will be to produce a set of types with the BNF description

```
Aty = A1 num | A2 num Bty
Bty = B1 Aty
```

The input to our mutually recursive types definition package is shown in Fig. 5. The structure `TypeInfo` defines `type_info`, which has the definition `datatype type_info = existing of hol_type | being_defined of string`.

We invoke our mutually recursive type definition package this way:

```
structure SampleDef =
  MutRecDefFunc
    (structure ExtraGeneralFunctions = ExtraGeneralFunctions
     structure MutRecTyInput = SampleInput
     structure SimpleDefineType = SimpleDefineType)
```

The result is that the HOL types `Aty` and `Bty` and the constructors `A1`, `A2`, and `B1` are defined. The contents of `SampleDef` are three theorems. The first is `New_Ty_Induct_Thm`:

```
 |- !Aty_Prop Bty_Prop.
      (!x1. Aty_Prop (A1 x1)) /\
      (!x1 x2. Bty_Prop x2 ==> Aty_Prop (A2 x1 x2)) /\
      (!x1. Aty_Prop x1 ==> Bty_Prop (B1 x1)) ==>
      (!x1. Aty_Prop x1) /\ (!x2. Bty_Prop x2)
```

```
val num = ==`:num`==;
structure SampleInput  : MutRecTyInputSig =
struct
structure TypeInfo = TypeInfo
open TypeInfo
val mut_rec_ty_spec =
[{type_name = "Aty",
  constructors =
  [{name = "A1", arg_info = [existing num]},
   {name = "A2", arg_info = [existing num, being_defined "Bty"]}]},
 {type_name = "Bty",
  constructors =
  [{name = "B1", arg_info = [being_defined "Aty"]}]}]
end (* struct *);
```

Fig. 5. Sample input

This theorem allows the user to prove inductive theorems about the members of the types by proving that the properties are true of the base cases (such as that for `A1`) and the inductive steps (such as that for `B1`).

The second is `New_Ty_Existence_Thm`:

```
  |- !A1_case A2_case B1_case.
       ?fn1 fn2.
         (!x1. fn1 (A1 x1) = A1_case x1) /\
         (!x1 x2. fn1 (A2 x1 x2) = A2_case (fn2 x2) x1 x2) /\
         (!x1. fn2 (B1 x1) = B1_case (fn1 x1) x1)
```

This allows the definition of mutually recursive functions. It is this theorem that is passed as an argument to `define_mutual_functions`.

The third is `New_Ty_Uniqueness_Thm`:

```
  |- ((!x1. fn1 (A1 x1) = A1_case x1) /\
      (!x1 x2. fn1 (A2 x1 x2) = A2_case (fn2 x2) x1 x2) /\
      (!x1. fn2 (B1 x1) = B1_case (fn1 x1) x1)) /\
     (!x1. fn1' (A1 x1) = A1_case x1) /\
     (!x1 x2. fn1' (A2 x1 x2) = A2_case (fn2' x2) x1 x2) /\
     (!x1. fn2' (B1 x1) = B1_case (fn1' x1) x1) ==>
     (fn1 = fn1') /\ (fn2 = fn2')
```

This theorem states that if two sets of mutually recursive functions are defined using `New_Ty_Existence_Thm`, then they are equal.

The SML function `define_mutual_functions` simplifies the definition of mutually recursive functions by allowing the user to simply describe the properties desired of the functions. As an example, we define the mutually recursive functions `count_A` and `count_B`:

```
val count_DEF =
  define_mutual_functions SampleDef.New_Ty_Existence_Thm
  (--`(count_A (A1 n) = n) /\
      (count_A (A2 n b) = n + (count_B b)) /\
      (count_B (B1 a) = (count_A a))`--);
```

The result is that the functions are defined, and the return value is a theorem that looks very much like the definition:

```
val count_DEF =
  |- (!x1. count_A (A1 x1) = x1) /\
     (!x1 x2. count_A (A2 x1 x2) = x1 + count_B x2) /\
     (!x1. count_B (B1 x1) = count_A x1) : thm
```

References

1. Stefan Kahrs. *Mistakes and Ambiguities in the Definition of Standard ML*, unpublished manuscript.
2. Robin Milner, Mads Tofte. *Commentary on Standard ML*, The MIT Press, Cambridge, Mass, 1991.
3. Robin Milner, Mads Tofte, Robert Harper. *The Definition of Standard ML*, The MIT Press, Cambridge, Mass, 1990.
4. Konrad Slind, Adding new rules to an LCF-style logic implementation. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 549 – 559. North-Holland, 1993.

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