Institute of Informatics, Warsaw University

The EML Kit
Version 1

Marcin Jurdziński
mju@mimuw.edu.pl

Mikołaj Konarski
mikon@mimuw.edu.pl

Aleksy Schubert
alx@mimuw.edu.pl

TR 96-04 (225)
May, 1996

Institute of Informatics
Warsaw University
ul. Banacha 2
02-097 Warsaw, Poland
phone: (48-22) 658-31-65
fax: (48-22) 658-31-64

Druk: Zakład Graficzny UW zam.nr. .

Contents

Preface

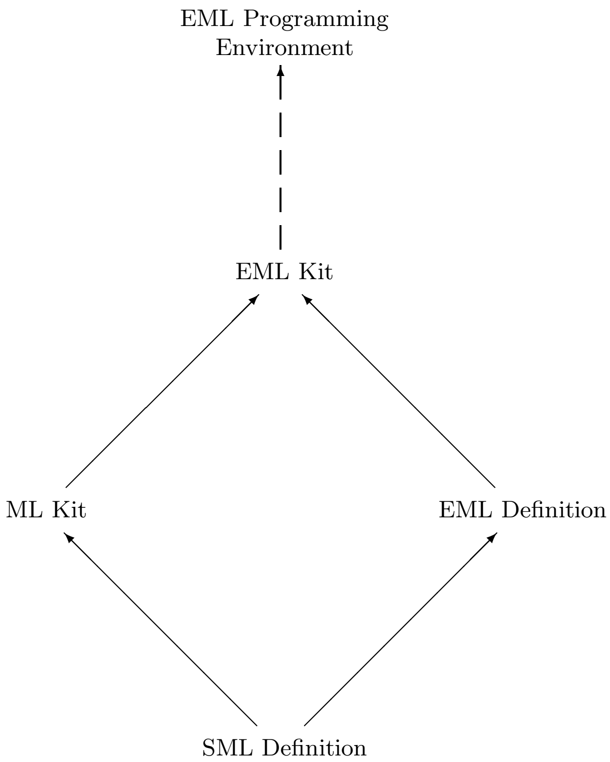
The EML Kit is an implementation of the Extended ML programming language.

Extended ML (EML) is a framework for the formal development of programs in the programming language Standard ML (SML). High-level specifications and SML code, as well as a mixture of both can be expressed in the Extended ML language. Thus the Extended ML language is an extension of (a large subset of) Standard ML.

Similarly the Definition of Extended ML [5] extends in a sense the Definition of Standard ML [8]. Although many things are added, some errors and infelicities corrected and a few presentation conventions changed, the general structure is kept mostly intact and as much of the original text as possible is preserved.

The EML Kit is based on the ML Kit [1]; a flexible, clean and modularized interpreter of the Standard ML, written in Standard ML. The ML Kit is a faithful and as straightforward as possible implementation of the language described in the SML definition. The overall structure of the ML Kit resembles the structure of the SML definition, abstract syntax tree is almost literally the same and the design of many details, such as naming conventions, is inspired by the Definition.

These two facts — that the EML definition is an extension of the SML definition and that the ML Kit is so close to the latter — helped us in our efforts to make the EML Kit a faithful and clean implementation of Extended ML. We were able to extend the ML Kit analogously as the EML definition extends the SML definition, while retaining the style of the ML Kit programming. Thus we have formed a basis for the development of the future EML Programming Environment tools, such as the proof-obligations generator and provers. See the figure below.



Moreover, it's possible that our work will be of use to people implementing the EML programming language on a different basis than the ML Kit. In particular our solutions to the problems arising when parsing EML programs or type-checking certain EML constructs can be of interest to them.

On the other hand, also people using the ML Kit as a basis for implementations of other programming languages which extend SML, may find some of our work useful. For example these fragments which deal with errors in the SML definition (which have their impact on the ML Kit) or these which were aimed at changing the ML Kit pretty-printing.

Acknowledgments

We would like to thank all the people who helped us during our work. First of all we would like to thank Andrzej Tarlecki who encouraged us to start the development of the EML Kit. Without his patient care and invaluable help we would never finish the task. We would also like to thank two other authors of the Extended ML: Don Sannella and Stefan Kahrs, who frequently offered us help and expressed interest in our work.

The EML Kit is based on the ML Kit, an SML interpreter praised at length in various parts of this document. We would like to express our gratitude to its authors: Lars Birkedal, Nick Rothwell, Mads Tofte and David N. Turner, for the months of work they've spent on making this excellent piece of a free software.

Robert Maron was the leader of our team during the early stages of the development of the EML Kit. We have not forgotten, Robert. Thank you.

Moreover thanks to the people from the Warsaw Applied Logic Group: Jerzy Tiuryn, Paweł Urzyczyn, Damian Niwiński, Michał Grabowski, Marek Zawadowski, Marcin Benke, Grzegorz Grudziński, Igor Walukiewicz, Jacek Chrząszcz, Stefan Dziembowski, Sławomir Leszczyński, Daria Walukiewicz and Adam Wierzbicki. We doubt the work we've made would ever be undertaken without the atmosphere they create here in Warsaw.

Part I
Introduction

The EML Kit is a parser, type-checker and evaluator of the Extended ML programming language. It has been developed for three purposes.

1 The EML Kit as a programming tool

The EML Kit can be used for type-checking and testing EML programs at all stages of their development.

The EML Kit can help in teaching EML or SML programming or in developing SML software with the EML formalism. In the latter case, if the efficiency of the resulting code is important, the EML Kit should be used in conjunction with a production quality SML compiler. An EML program should be transformed, with the help of the EML Kit, from its initial high-level EML form, through all the intermediate EML stages, to the final SML code. Then the SML code should be compiled by the SML compiler.

The executable called `eml`, which parses, type-checks and evaluates EML programs, seems to be the most suitable for people using the EML Kit as a programming tool. The user interface of this executable is similar to the user interface of the well known SML/NJ compiler. The `use` primitive is accessible at the the top-level, allowing to import files in the same way as in most SML compilers.

2 The EML Kit as a vehicle for experiments

The EML Kit lets the EML authors and all the people involved in the EML-related work fiddle with a straightforward implementation of the EML language.

Current research concerning EML is centered on the Verification Semantics of EML and on issues which one may call "implementation" of the Verification Semantics. These include designing a proof theory for EML and building prototype provers. The Verification Semantics heavily uses the Static Semantics, e.g. for expressing the property of having a type. Moreover the Static Semantics of the EML is burdened with the responsibility of gathering various information for the Verification Semantics. The EML Kit contains an implementation of the Static Semantics of EML, including the "gathering" mechanisms.

For the people treating the EML Kit as a medium of experiments, the best tool is the `emlkit` executable. Its user interface is very similar to the ML Kit user interface, which is described best in the ML Kit documentation [1]. In short, the `emlkit` consists of an SML interpreter with the initial basis enriched by types and functions implementing EML definition's semantic objects, rules, etc. Among others there are functions `parse`, `elab`, `eval` and `parseFile`, `elabFile`, `evalFile`, which can be used for respectively; parsing, elaborating or evaluating of the EML programs in the interactive or non-interactive way.

3 The EML Kit as a "denotational definition"

The EML Kit can be thought of (with some amount of good will) as a fully deterministic, algorithmic and detailed description of the EML static and dynamic semantics.

The EML definition describes the EML language using a kind of BNF notation for its grammar, and an extension of the formalism called Natural Semantics for the Static and Dynamic Semantics. This form of presentation makes it possible to describe such a large system as EML in a relatively compact and clean way. There is a price to pay, however. Some small mistakes long remain unnoticed. Some mechanisms seem simple and unambiguous, but the first attempt to implement them shows that they are not. Some global relations between parts of the Definition are obscure until the modules hierarchy is described in a serious module-handling formalism, like the one of the SML programming language.

We hope that this report will describe results of the analysis of EML from the perspective of the EML Kit, as well as give an overview of the EML Kit seen as an EML "denotational definition". We reckon, that a knowledge of at least the Definition of Extended ML [5] is essential for the understanding of our paper.

Part II of this paper, written by Aleksy Schubert, describes the changes to the ML Kit parsing, which were needed to obtain the implementation of the Extended ML Full Grammar and Derived Forms. Part III, written by Mikołaj Konarski, is devoted to issues related with the implementation of the Static Semantics of EML. Part IV, written by Marcin Jurdziński, describes the implementation of the Dynamic Semantics of EML and the design of several small but important parts of the EML Kit.

Part II
Parsing

The present parsing for the EML Kit implements the complete full grammar of EML and its syntactic restrictions as stated in the Definition of Extended ML except for one rule. The rule for *strdec* looks like this:

$$
\begin{array}{rl}
\textit{strdec} ::= & \textit{dec} \\
& \texttt{axiom}\ \textit{ax} \\
& \qquad \langle \texttt{withtype}\ \textit{typbind}^{\bullet} \rangle \\
& \ldots
\end{array}
$$

in the book. Some discussions, however, led us and EML designers to conclusion that ⟨`withtype` *typbind*$^{\bullet}$⟩ was an error here.

In addition to the grammar of the EML definition we kept some features of the original ML Kit grammar, especially pertinent to imperative features, for instance `while` statements.

4 Major changes to the parsing in the ML Kit

Most of the changes we made were very straightforward. They consisted in simply implementing the tree generation for suitable rule of full grammar. There were however three bigger changes made to the parsing:

- introducing bullets to some rules,
- making the rule for *specexp* work,
- syntactic restriction of guarded *strexp* in a structure binding.

In the EML definition, bullets were used to indicate in some places that question marks are not allowed. As far as in one place bullet was added to nonterminal *exp* it is more or less clear why the first change could not be so small. The troubles in the second case were caused by deep ambiguity in the full grammar of EML as presented in the EML definition. The problem was that in the rule:

$$
\begin{array}{rl}
\textit{specexp} ::= & \texttt{let}\ \textit{strdec}\ \texttt{in}\ \textit{axexp}\ \texttt{end} \\
& \textit{exp}^{\bullet}
\end{array}
$$

the same program might be parsed according to both the first, and the second line of the production because it was possible in *strdec* to have usual *dec* so that in some cases the first line might reduce to the following one:

$$\texttt{let}\ \textit{dec}\ \texttt{in}\ \textit{axexp}\ \texttt{end}$$

which is very similar to one of the rules of *exp*$^{\bullet}$. Such ambiguity manifested itself for instance in the following EML specification:

```
signature S =
sig
 axiom let val x = 5 in x > x end
end;
```

The third major change was simpler. It consisted in introducing additional nonterminal for guarded *strexps*.

Now we are going to describe the three changes in more detail.

4.1 Introducing bullets

We were considering two ways of introducing bullets:

- by an additional check after yacc parsing; in the fashion the problem of infixing was solved in the ML Kit,
- directly in the yacc–grammar.

There were several pros and cons for both methods.

The additional check seemed to be simpler to implement in the sense that the situation of the programmer was clear. He was to make it by himself from the beginning to the end and he did not have to analyze complicated yacc parsing of the ML Kit. Such solution was not too good because it was clear that bullets might be implemented through yacc parsing, and that they were not as difficult as the infixing problem (infixing makes the ML grammar not context–free).

The yacc parsing method seemed to be difficult because of plenty of rules which had to be considered during the implementation. Consequently the method might lead to a greater number of bugs in the EML Kit than the other one.

We decided to adopt the latter method because it seemed to us to be more regular. Consequently the general structure of the ML Kit was left unchanged (which would not be the case in the solution of additional post-parse stage). The risk of introducing bugs did not seem very significant to us, since most of the new rules would be generated simply by copying parts of the existing ones.

Changes in details. Now we jump to more detailed description of the changes we made.

There are three nonterminals with bullets in the full grammar of the EML definition:

— *exp*

— *match*

— *typbind*

We introduced for each of these nonterminals two new nonterminals in the EML Kit grammar:

```
Exp_bullet,
Exp_questionmark,
Match_bullet,
Match_questionmark
TypBindbullet,
TypBindquestionmark
```

respectively. Nonterminals with bullet in their name generate parts of EML program that do not involve question marks, and nonterminals with `questionmark` in their name generate parts that do have such sign.

We eliminated the original ML Kit nonterminals `TypBind`, and `Match_` because they were needed only in few productions, and we kept the nonterminal `Exp_` because of great number of places it occurred. Moreover when we tried to define `TypBind` and `Match_` as a sum of `questionmark` and `bullet` parts we encountered certain troubles with shift/reduce conflicts.

Now we are going to present fragments of rules for *exp* as an illustrating example for changes we made. At the moment, rules look as follows:

The rule below gives sum of `bullets` and `questionmarks` for `Exp_`:

```
Exp_:     Exp_questionmark ( Exp_questionmark )
      |   Exp_bullet ( Exp_bullet )
```

This kind of rule is not present in `Match_` or `TypBind`.

Here are rules for question marks. This example illustrates how binary operations are handled in the rules with `questionmark`. We simply build expressions that have question mark in the left, the right, and in both branches of an expression. The presence of bulleted nonterminal is necessary to avoid ambiguity:

```
Exp_questionmark: ...
        | Exp_questionmark HANDLE Match_bullet
                  ( HANDLEexp(PP Exp_questionmarkleft Match_bulletright,
                              Exp_questionmark, Match_bullet) )
        | Exp_bullet HANDLE Match_questionmark
                  ( HANDLEexp(PP Exp_bulletleft Match_questionmarkright,
                              Exp_bullet, Match_questionmark) )
        | Exp_questionmark HANDLE Match_questionmark
                  ( HANDLEexp(PP Exp_questionmarkleft
                                 Match_questionmarkright,
                              Exp_questionmark, Match_questionmark) )
        | ...
```

In the rule for `Exp_questionmark`, we do not have cases for quantifiers, comparison (`==`) and the convergence predicate because all these rules would forbid occurrence of question mark which is required in the rule for `Exp_questionmark`.

In the case of `Exp_bullet`, we do not have to introduce so many rules for binary operations:

```
Exp_bullet: ...
        | Exp_bullet HANDLE Match_bullet
             ( HANDLEexp(PP Exp_bulletleft Match_bulletright,
                         Exp_bullet, Match_bullet))
        | ...
```

is all we need for the `handle` operation. Obviously we introduce rules for quantifiers, comparison (`==`) and convergence predicate:

```
Exp_bullet: ...
        | EXISTS Match_bullet
             ( EXIST_QUANTexp(PP EXISTSleft Match_bulletright,
                              Match_bullet) )
```

```
| FORALL Match_bullet
     ( UNIV_QUANTexp(PP FORALLleft Match_bulletright,
                     Match_bullet) )
| Exp_bullet TERMINATES
     ( CONVERexp(PP Exp_bulletleft TERMINATESright,
                Exp_bullet ) )
| Exp_bullet EQUALEQUAL Exp_bullet
     ( COMPARexp(PP Exp_bullet1left Exp_bullet2right,
                Exp_bullet1, Exp_bullet2) )
| Exp_bullet EQUALSLASHEQUAL Exp_bullet
     ( ifThenElse( (COMPARexp(PP Exp_bullet1left
                               Exp_bullet2right,
                            Exp_bullet1, Exp_bullet2)),
                  falseExp, trueExp))
| ...
```

The changes described above led to another set of fresh nonterminals for the yacc–grammar; the new set led to another one; and so on. A closure was reached at the following set of rules for bullets:

```
| BarMatch_optbullet           of match Option
| MRulebullet                  of mrule
| ValBindbullet                of valbind
| AndValBind_optbullet         of valbind Option
| AndFnValBind_optbullet       of valbind Option
| FnValBindbullet              of valbind
| AndTypBind_optbullet         of typbind Option
| OneDec_sans_LOCALbullet      of dec  (* One DEC phrase, no LOCAL. *)
| OneDecbullet                 of dec
| OneDec_or_SEMICOLONbullet    of dec Option
| NonEmptyDecbullet            of dec
| Decbullet                    of dec  (* One DEC phrase. *)
| AtExpbullet                  of atexp
| AtExp_seq1bullet             of atexp list
| ExpRowbullet                 of exprow
| ExpRow_optbullet             of exprow Option
| CommaExpRow_optbullet        of exprow Option
| ExpComma_seq0bullet          of exp list
| ExpComma_seq1bullet          of exp list
| ExpComma_seq2bullet          of exp list
| ExpSemicolon_seq2bullet      of exp list
| Exp_bullet                   of exp
| Match_bullet                 of match
| FValBindbullet               of FValBind
| AndFValBind_optbullet        of FValBind Option
| FClausebullet                of FClause
| BarFClause_optbullet         of FClause Option
| TypBindbullet                of typbind
```

and the following one for question marks:

```
| Match_questionmark           of match
| BarMatch_optquestionmark     of match Option
| MRulequestionmark            of mrule
```

```
| Exp_questionmark              of exp
| AtExp_seq1questionmark        of atexp list
| AtExpquestionmark             of atexp
| ExpRowquestionmark            of exprow
| ExpRow_optquestionmark        of exprow Option
| CommaExpRow_optquestionmark   of exprow Option
| ExpComma_seq0questionmark     of exp list
| ExpComma_seq1questionmark     of exp list
| ExpComma_seq2questionmark     of exp list
| ExpSemicolon_seq2questionmark of exp list
| FValBindquestionmark          of FValBind
| AndFValBind_optquestionmark   of FValBind Option
| FClausequestionmark           of FClause
| BarFClause_optquestionmark    of FClause Option
| OneDec_sans_LOCALquestionmark of dec  (* One DEC phrase, no LOCAL. *)
| OneDecquestionmark            of dec  (* One DEC phrase. *)
| OneDec_or_SEMICOLONquestionmark  of dec Option
| NonEmptyDecquestionmark       of dec
| Decquestionmark               of dec
| TypBindquestionmark           of typbind
| AndTypBind_optquestionmark    of typbind Option
| ValBindquestionmark           of valbind
| FnValBindquestionmark         of valbind
| AndValBind_optquestionmark    of valbind Option
| AndFnValBind_optquestionmark  of valbind Option
```

Analogously we ruled out some nonterminals handled by nonterminals with `bullets` and `questionmarks` in names. It is worth remarking that introducing these nonterminals via simple sum rules (like in the first example for `Exp`) either caused ambiguities or simply was not needed because they disappeared during the process of producing nonterminals with `bullets` and `questionmarks` in names.

```
| TypBind                       of typbind
| ValBind                       of valbind
| FnValBind                     of valbind
                                        (* RHS of ‘val rec’ binding. *)
| FValBind                      of FValBind
                                        (* ‘fun’ binding. *)
| Dec                           of dec
| MRule                         of mrule
| Match_                        of match
                                        (* Exp_, Match_ rather than *)
                                        (* Exp, Match which are the *)
                                        (* pervasive exceptions... *)
| ExpRow                        of exprow
| AtExp                         of atexp
| FClause                       of FClause
| CommaExpRow_opt               of exprow Option
| AndValBind_opt                of valbind Option
| AndFnValBind_opt              of valbind Option
| AndFValBind_opt               of FValBind Option
| BarFClause_opt                of FClause Option
```

```
| AndTypBind_opt              of typbind Option
| BarMatch_opt                of match Option
| ExpRow_opt                  of exprow Option
| ExpComma_seq0               of exp list
| ExpComma_seq1               of exp list
| ExpComma_seq2               of exp list
| AtExp_seq1                  of atexp list
| ExpSemicolon_seq2           of exp list
| OneDec_or_SEMICOLON         of dec Option
| NonEmptyDec                 of dec
| OneDec                      of dec  (* One DEC phrase. *)
```

4.2 Introducing `let` to *specexp*

The rule for *specexp* is more complex to handle than other ones, as illustrated by the example at the beginning of the Section 4.

We are going to describe intuitively the solution we chose. We should have solved ambiguity arising from the rule. There were two ways to do this:

- to parse all the strings that look like *exp* as if they were *exp*, or
- to parse all the `lets` as if they were `lets` with *strdec*.

We took the first solution and decided that all the *specexps* that looked like *exps* had to be treated like *exps*. This allowed us not to change the rule for `Exp_bullet`, which seemed to be complicated. Such change would be very difficult to make because it would require for instance `let` in expressions like:

```
let val x = 6 in x end + 3
```

to be parsed as *exp* and not as some kind of `let` *strdec* ..., what would take place without the expected tedious changes.

Our decision led to three new nonterminals:

```
| StrDec_SpecExp              of strdec
| OneStrDec_SpecExp           of strdec
| NonEmptyStrDec_SpecExp      of strdec
```

They are used to keep the track of *strdecs* that are not *decs*. The implementation is similar as in the case of nonterminals used to handle question marks:

```
NonEmptyStrDec_SpecExp:
          NonEmptyStrDec_SpecExp OneStrDec_or_SEMICOLON
                        ( case OneStrDec_or_SEMICOLON
                              of Some strdec =>
                                   composeStrDec(PP NonEmptyStrDec_SpecExpleft
                                                    OneStrDec_or_SEMICOLONright,
                                   NonEmptyStrDec_SpecExp, strdec)
                               | None =>
                                   NonEmptyStrDec_SpecExp )

        | OneStrDec_SpecExp
                        ( OneStrDec_SpecExp)
```

```
        | NonEmptyDecbullet OneStrDec_SpecExp
                        ( composeStrDec(
                                   PP NonEmptyDecbulletleft
                                      OneStrDec_SpecExpright,
                                   Dec2StrDec( PP NonEmptyDecbulletleft
                                                  NonEmptyDecbulletright,
                                               NonEmptyDecbullet ),
                                   OneStrDec_SpecExp) )
```

4.3 Guarded *strexps*

We introduced one new nonterminal `StrExp_guarded`. According to the last syntactic restriction from p. 17 of the EML definition we introduced the following generation rules for the new nonterminal:

```
 StrExp_guarded:
          LongIdent
                        ( LONGSTRIDstrexp(PP LongIdentleft LongIdentright,
                                          mk_LongStrId LongIdent
                                         )
                        )
        | Ident LPAREN StrExp RPAREN
                        ( APPstrexp(PP Identleft RPARENright,
                                    mk_FunId Ident, StrExp
                                   )
                        )
        | LET StrDec IN StrExp_guarded END
                        ( LETstrexp(PP LETleft ENDright, StrDec, StrExp) )
```

Of course, we had to introduce a suitable rule for the single structure binding generation:

```
SglStrBind: ...
    | Ident EQUALS StrExp_guarded
          ( UNGUARDsglstrbind(PP Identleft StrExp_guardedright,
                              mk_StrId Ident, StrExp_guarded))
```

5 Minor changes to the parsing in the ML Kit

Now we are going to describe minor changes we made to the ML Kit.

5.1 Changes in the lexing

The file `Topdec.lex` was left untouched. We added some new cases to the `keyword` function in the file `LexUtils.sml`:

```
   fun keyword tok = (shifting("KEY(" ^ text ^ ")"); tok(p1, p2))
      in
        case text of ...
      | "axiom"     => keyword AXIOM
      | "exists" => keyword EXISTS
      | "forall" => keyword FORALL
      | "implies"   => keyword IMPLIES
```

```
| "proper" => keyword PROPER
| "raises" => keyword RAISES
| "terminates" => keyword TERMINATES

| "=="    => keyword EQUALEQUAL
| "=/="   => keyword EQUALSLASHEQUAL
| "?" => keyword QUESTIONMARK
```

These are all the new EML keywords[1].

5.2 Changes in yacc–parsing

5.2.1 The treatment of new reserved words

We added new reserved words to the grammar

- as terminals:

```
%term ...
   | AXIOM
   | IMPLIES
   | EXISTS | FORALL | PROPER | RAISES | TERMINATES
   | QUESTIONMARK | EQUALEQUAL | EQUALSLASHEQUAL
```

- and as keywords:

```
%keyword EQTYPE FUNCTOR INCLUDE SHARING SIG SIGNATURE STRUCT ...
      AXIOM
      IMPLIES
      EXISTS FORALL PROPER RAISES TERMINATES QUESTIONMARK
      EQUALEQUAL EQUALSLASHEQUAL
```

The associativity declaration for new reserved words is:

```
%right   IMPLIES
%right EXISTS
%right FORALL (* quantifiers seem to behave like IMPLIES *)
%left PROPER
%left RAISES
%left TERMINATES
%right EQUALEQUAL
%right EQUALSLASHEQUAL
```

5.2.2 The treatment of nonterminals

We added plenty of new nonterminals. In general we added new nonterminals where the grammar did (for instance for *psigexp* is `PSigExp` or *specexp* is `SpecExp`), and where it was necessary from the point of view of yacc (for instance `AndAxDesc_opt`). Here is the list of the nonterminals we added following the grammar (p. 119 of the EML definition; we omit the nonterminals introduced due to bullets):

[1] The keyword `implies` was added by Robert Maron on the early stages of the work on the EML Kit.

```
| AxExp          of axexp
| Ax             of ax
| SglStrBind     of sglstrbind
| PSigExp        of psigexp
| AxDesc         of axdesc
| SpecExp        of specexp
```

and here is the list of the nonterminals we added due to yacc requirements:

```
| AndAxDesc_opt     of axdesc Option
| AndAx_opt         of ax Option
```

The latter list is so modest because most of the new nonterminals added due to yacc requirements are either with `questionmarks` or `bullets`.

How nonterminals of the EML definition grammar are handled by the EML Kit parsing. We are going to present a handful of tables examining one by one the way rules from pp. 119–124 of EML definition are handled in the EML Kit. The tables are organized in the same fashion the tables of the EML Kit grammar are.

Table 1: **Nonterminals for expressions and matches**

nonterminal	*way of handling*
atexp	handled by `AtExpbullet` and `AtExpquestionmark`
exprow	handled by `ExpRowbullet` and `ExpRowquestionmark`
appexp	handled by `AtExp_seq1bullet` and `AtExp_seq1questionmark` and then by infixing
infexp	it is parsed as *appexp* and then resolved by infixing
exp	handled by `Exp_bullet` and `Exp_questionmark`, there is also production for `Exp_` too
match	`Match_bullet` and `Match_questionmark`
mrule	`MRule_bullet` and `MRulequestionmark`

Table 2: **Nonterminals for declarations and bindings**

nonterminal	*way of handling*
dec	`Decbullet` and `Decquestionmark`[2]
valbind	`ValBindbullet` and `ValBindquestionmark`
fvalbind	`FValBindbullet` and `FValBindquestionmark`
funcbind	`FClausebullet` and `FClausequestionmark`
typbind	`TypBindbullet` and `TypBindquestionmark`

[2]The way it is handled is more complicated; this includes non empty *decs* and so on.

nonterminal	*way of handling*
datbind	`DatBind`
conbind	`ConBind`
exbind	`ExBind`

Table 3: **Nonterminals for patterns**

nonterminal	*way of handling*
atpat	`AtPat`
patrow	`PatRow`
apppat	analogously to *appexp* handled by `AtPat_seq2` and the infixing stage
infpat	it is parsed as *apppat* and then resolved by infixing
pat	`Pat`
fpat	`AtPat_seq1`

Table 4: **Nonterminals for type expressions**

nonterminal	*way of handling*
ty	`Ty`
tyrow	`TyRow`

Table 5: **Nonterminals for structure and signature expressions**

nonterminal	*way of handling*
strexp	`StrExp`
strdec	`StrDec`[3]
ax	`Ax`
axexp	`AxExp`
strbind	`StrBind`
sglstrbind	`SglStrBind`
sigexp	`SigExp`
psigexp	`PSigExp`
sigdec	`SigDec_sans_SEMICOLON`
sigbind	`SigBind`

[3]The way it is handled is more complicated; this includes non empty *strdecs* and so on.

Table 6: **Nonterminals for specifications**

nonterminal	*way of handling*
spec	`Spec`
valdesc	`ValDesc`
typdesc	`TypDesc`
datdesc	`DatDesc`
condesc	`ConDesc`
exdesc	`ExDesc`
specexp	`SpecExp` — discussed in Section 4.2
strdesc	`StrDesc`
shareq	`SharEq`

Table 7: **Nonterminals for functors and top–level declarations**

nonterminal	*way of handling*
fundec	`FunDec_sans_SEMICOLON`
funbind	`FunBind`
topdec	`TopDec`
program	it is handled by the `%eop SEMICOLON EOF` declaration in the header of `Topdec.grm` file
fullprogram	in the light of the situation in *program*, it is not necessary

We eliminated some nonterminals too for instance `ColonSigExp_opt` because `ColonSigExp` is no longer optional.

We had to add some new functions to `GrammarUtils.sml` file to help to build abstract syntax tree. These functions are:

Table 8: **Functions that help to build abstract syntax tree**

function	*What the function does.*
`mk_PSigId`	creates a principal signature expression out of an identifier
`specAsSig`	creates a principal signature expression out of specification
`Dec2StrDec`	translates ordinary declaration to a structure declaration — it is used in parsing *specexps*
`functSigExp`	it had already existed and was changed to handle rather principal *sigexps*
`raisesExp`	creates `HANDLEexp` suitable for expression with `raises`

function	What the function does.
`properExp`	creates `HANDLEexp` suitable for expression with `proper`

It is worth mentioning that the `lookup_tycon` function has been enriched with cases handling the question mark binding.

We spare the Reader further boring details of the implementation. Of course all the details are available directly from the authors.

5.3 Changes in the infixing

Now we are going to describe things connected with the changes in the infixing. The purpose of most of the changes was to introduce new resolving functions or to introduce new patterns to existing functions for new kinds of nodes of abstract syntax tree. Here is the list of — hopefully self-explanatory — names of added resolving functions[4]:

```
resolveSigdec
resolveAX
resolveSigexp
resolveStrDesc
resolveAxDesc
resolveSpecExp
resolveAxExp
resolveFunbind
resolveStrbind
```

We added the following new patterns to the existing functions:

- `AXIOMstrdec` to `resolveStrDec` (new node),
- `AXIOMspec` and `STRUCTUREspec` to `resolveSpec` (new node `AXIOMspec`, and structures may include expressions in axioms now),
- `EQTYPEdec` in `resolveDec` (new node),
- `COMPARexp`, `EXIST_QUANTexp`, `UNIV_QUANTexp` and `CONVERexp` to `resolveExp` (new nodes),
- `UNDEFatexp` to `resolveAtExp` (new node).

5.4 Changes to interfaces of structures involved in parsing

It came out that some sharing equations for *exps* are needed in the interfaces. They are introduced in the `GrammarUtils.sml` and the `Parse.sml` files. In both cases, the added sharing equation is[5]:

```
and type TopdecGrammar.exp = DecGrammar.exp
```

[4]Functions `resolveSigdec` and `resolveSigexp` are due to Robert Maron.

[5]These changes are due to Mikolaj Konarski.

Part III
Elaboration

6 Static Semantics of Extended ML

The Definition of Extended ML uses a formalism called Natural Semantics (a kind of operational semantics) to describe the static semantics of EML. There is a collection of semantic objects, defined using mathematical notation, and a set of rules to be used in derivation of judgments of the form

$$C \vdash \mathit{phrase} \Rightarrow A$$

which can be read "*phrase* elaborates to A in context C" (C, A are semantic objects, *phrase* is an EML phrase).

Fundamental concepts of the Static Semantics are defined using quantification over derivations. For example *phrase* is said to possess a type τ in a context C, if there *exists* a derivation of the sentence

$$C \vdash \mathit{phrase} \Rightarrow \tau$$

In addition some of the conditions in the rules express properties of the form "*for all* derivations such that ϕ holds, ψ must hold" — good examples are the principality requirements appearing in the Static Semantics for Modules.

Because of the quantification over infinite sets, it is impossible to implement Static Semantics of EML literally. One has to translate it into a more "denotational" form (some prefer to call it a "deterministic" form) before it can be straightforwardly (at last) implemented in a programming language like SML. Such a reformulation is a non-trivial task.

Fortunately the similar problem concerning SML has been solved quite a while ago. Milner's algorithm [2] is a well known skeleton for an SML Core language type-checker, and the proof of the Principality Theorem from Section A.2 of the Commentary on Standard ML [7] is a good guideline for dealing with SML Modules language. Last but not least, the ML Kit is a very clean and modest reformulation/implementation of the SML Static Semantics, based on the above-mentioned ideas.

Our work on reformulating the EML-specific fragments of the Static Semantics sometimes amounted to trivial extensions of e.g. Milner's algorithm, the only work being a struggle with the ML Kit machinery. Sometimes however the extension required a proof of correctness, or a very good knowledge of the ML Kit design. And still sometimes a new approach had to be found, proven correct and implemented within the ML Kit frames.

The greater part of our work was being done after the Version 1 of the EML definition [5] had been published. In the result of our work and our discussions with the EML authors we found out that some changes to the definition would simplify the task of its implementation, or make the definition as seen from the perspective of its deterministic reformulation, more elegant. The authors of Extended ML have agreed to incorporate these changes into the future version of the definition. Meanwhile we are going to use the Definition of Extended ML, Version 1 as basis for the discussion of the elaboration part of the EML Kit, referring when necessary to the proposed modifications, described in detail in Section 11.

7 Implementation of EML-specific language constructs

Even those marginally familiar with the EML expect, that to change an SML compiler into an EML compiler one has to add some code for handling new EML keywords like `forall`, `terminates`, etc.

7.1 New Core language constructs

There are several rules in the EML Static Semantics for the Core, with no counterparts in the SML definition. These rules describe the typing of the new EML Core language constructs. They are listed in Table 9.

Table 9: **The rules describing the typing of the new Core language constructs**

rule number	*form of the construct*	*name of the construct*	*phrase class*
7.1	`?`	undefined value	*atexp*
11.1	$exp_1^\bullet$ `==` $exp_2^\bullet$	comparison	*exp*
11.2	`exists` $match^\bullet$	existential quantifier	*exp*
11.3	`forall` $match^\bullet$	universal quantifier	*exp*
11.4	$exp^\bullet$ `terminates`	convergence predicate	*exp*
18.1	`eqtype` *typbind*	equality type declaration	*dec*
28.1	*tyvarseq tycon* `= ?`	question mark type binding	*typbind*

Let's take a closer look at the simplest one:

$$\frac{C \vdash exp^\bullet \Rightarrow \tau, U, \gamma}{C \vdash exp^\bullet\ \mathtt{terminates} \Rightarrow \mathtt{bool}, U, \epsilon} \tag{11.4}$$

This can be read: "if $exp^\bullet$ can elaborate in the context C to the type τ, set U and trace γ then $exp^\bullet$ `terminates` can elaborate in the context C to the type `bool`, set U and empty trace". For the purpose of implementation we can read it "if $exp^\bullet$ elaborates in the context C to its most general type τ, set U and its most general trace γ then $exp^\bullet$ `terminates` elaborates in the context C to the type `bool`, set U and empty trace". This reformulation can be justified by an easy proof that Milner's algorithms extended to cover this rule remains correct.

Here is the implementation of the rule:

```
fun elab_exp (C, exp) =
  case exp of
  ...
  (* Convergence predicate expression *)                    (* rule 11.4 *)
     | IG.CONVERexp(i, exp) =>
         let
           val (S, tau, out_exp) = elab_exp(C, exp)
         in
           (S, StatObject.TypeBool, OG.CONVERexp(okConv i, out_exp))
         end
```

`CONVERexp` is the name of the abstract syntax tree node (see Section 12) representing the `terminates` construct. `i` is an info item used to propagate error messages and other information during elaboration. In this case it's converted using `okConv`, to indicate that the elaboration was successful in this particular point. S is a substitution needed by the machinery of Milner's algorithm, The code looks quite straightforward, partly because the U set and the trace are taken care of elsewhere.

The implementation of other rules is not much more conceptually complicated, although expressing some of the side conditions, such as the one from rule 18.1:

$$\forall(\theta, CE) \in \mathrm{Ran}\, TE,\ \theta \text{ admits equality}$$

which from a mathematical point of view looks very innocently, required a lot of struggle with the implementation of the Static Objects.

7.2 Axioms

Axioms can appear in two different places — in structures (rule 57.1) and in signatures (rule 74.1). When appearing inside structures axioms may have quite a complicated body, e.g.:

```
axiom exists i => 1 - i == 1 + i
  and ((fn x => x) 3) terminates
```

Inside signatures they can be even more complex, e.g.:

```
axiom let
          structure Order : ORDER = ?
      in
          forall (x, y) => (Order.leq (x, y)) proper
      end
  and (fn x => x) == (fn y => y)
```

The rules describing the semantics of various possible forms of axiom bodies are listed in Table 10.

Table 10: **The rules describing the semantics of various possible forms of axiom bodies**

rule number	*form*	*phrase class*
57.1	`axiom` *ax*	*strdec*
61.1	*axexp* ⟨`and` *ax*⟩	*ax*
61.2	$exp^\bullet$	*axexp*
74.1	`axiom` *axdesc*	*spec*
86.2	*specexp* ⟨`and` *axdesc*⟩	*axdesc*
86.3	`let` *strdec* `in` *axexp* `end`	*specexp*

The implementation of all of these rules except rule 61.2 was quite easy. Rule 61.2 defines the elaboration of the simplest form of an axiom body: $exp^\bullet$ — an expression with no question marks allowed inside. Modified according to our suggestions (see Sections 11.2 and 11.3), rule 61.2 looks as follows:

$$\frac{C \text{ of } B \vdash exp^\bullet \Rightarrow \texttt{bool}, \emptyset, \gamma \qquad \dfrac{C \text{ of } B \vdash exp^\bullet \Rightarrow \tau, \emptyset, \gamma'}{\mathrm{Clos}\gamma \succ \gamma' \wedge \tau = \texttt{bool}}}{B \vdash exp^\bullet \Rightarrow \mathrm{Clos}\gamma} \tag{61.2}$$

The implementation of this rule was quite difficult and complicated. A major problem was, that a trace collected at one of the intermediate stages of the $exp^\bullet$ elaboration cannot be closed with respect to context of this stage, because the context needn't be principal yet. We had to design an involved algorithm to correctly organize the ongoing collection, updating and closure of traces during elaboration. The simplified version of the skeleton of the implementation of rule 61.2 is given below. The `ElabDec` module referred to frequently in this piece of code is responsible for a large part of the Core language elaboration, and contains (by our decision) most of our tools for elaboration of axioms.

```
(**************************************************)
(* Axiomatic expressions - Definition (eml) page 47 *)
(**************************************************)

fun elab_axexp (B : Env.Basis, axexp : IG.axexp) : OG.axexp =
    case axexp of
         IG.AXIOM_EXPaxexp(i, exp : IG.exp) =>
           let
               val (tau, out_exp) =
                   ElabDec.elab_resolve_close_and_process_exp(
                   Env.C_of_B B, exp)
           in
               if tau = ElabDec.TypeBool then
                   if ElabDec.U_of_exp_empty(exp) then
                       OG.AXIOM_EXPaxexp(okConv i, out_exp)
                   else OG.AXIOM_EXPaxexp(errorConv(i,
                       ErrorInfo.UNGUARD_EXPLICIT_TV_IN_AXEXP), out_exp)
               else OG.AXIOM_EXPaxexp(errorConv(i,
                   ErrorInfo.AXEXP_SHOULD_BE_BOOL), out_exp)
           end
```

8 Implementation of the EML module system

8.1 Structure bindings

In the Definition of Standard ML rule 62 was describing completely the elaboration of structure bindings. In the Definition of Extended ML the abstract syntax tree has changed due to the introduction of undefined structure bindings and the restrictions concerning guardedness. In the result, rule 62:

$$\frac{B \vdash \mathit{sglstrbind} \Rightarrow \mathit{SE}, \gamma \qquad \langle B + \mathrm{names}\, \mathit{SE} \vdash \mathit{strbind} \Rightarrow \mathit{SE}', \gamma' \rangle}{B \vdash \mathit{sglstrbind}\ \langle \mathtt{and}\ \mathit{strbind} \rangle \Rightarrow \mathit{SE}\ \langle + \mathit{SE}' \rangle, \gamma \langle \cdot \gamma' \rangle} \tag{62}$$

serves only for handling the optional components of structure bindings. The task of defining the semantics of structure/signature matching and related issues falls now to the rules describing the elaboration of single structure bindings.

8.2 Single structure bindings

There are two very important and new features of the EML module system. First, modules behave like abstractions, which means that the semantics of a module is fully determined by its signature. Second, undefined structure and functor bindings are introduced. In effect, one can declare a module and safely refer to it even before the implementation of this module is written.

The EML rules for single structure bindings express the semantics of structures-abstractions, of undefined structure bindings and of unguarded structure bindings without too much verbosity, in a unified, non-conflicting way:

$$\frac{\begin{array}{cc} B \vdash \mathit{psigexp} \Rightarrow (N)S, \gamma & B \vdash \mathit{strexp} \Rightarrow S', \gamma' \\ N \cap N \text{ of } B = \emptyset & (N)S \geq S'' \prec S' \end{array}}{B \vdash \mathit{strid} : \mathit{psigexp} = \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto S\},\ \gamma \cdot \gamma'} \tag{62.1}$$

$$\frac{B \vdash \mathit{psigexp} \Rightarrow (N)S, \gamma \qquad N \cap N \text{ of } B = \emptyset}{B \vdash \mathit{strid} : \mathit{psigexp} = \texttt{?} \Rightarrow \{\mathit{strid} \mapsto S\},\ \gamma} \tag{62.2}$$

$$\frac{B \vdash \mathit{strexp} \Rightarrow S, \gamma}{B \vdash \mathit{strid} = \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto S\},\ \gamma} \tag{62.3}$$

Note that the third premise in rule 62.1 is always satisfied in the implementation, because at the beginning of elaboration of *psigexp* the free names are chosen to be distinct from the names of the basis, and in the meantime no alpha-conversion of the result is made.

Here is the fragment of the EML Kit source code directly corresponding to these rules:

```
(*****************************************************)
(* Single Structure Bindings - Definition (eml) page 47 *)
(*****************************************************)

fun elab_sglstrbind (B: Env.Basis, sglstrbind: IG.sglstrbind):
  (Env.StrEnv * OG.sglstrbind) =

  case sglstrbind of

   (* Single structure bindings *)
   IG.SINGLEsglstrbind(i, strid, psigexp, strexp) =>
    let
      val (S', out_strexp) = elab_strexp(B, strexp)
      val (Sigma, out_psigexp) = elab_psigexp(B, psigexp)
      val (i', S'')  = sigMatchStr(i, Sigma, S')
      val (N, S) = Stat.unSig(Sigma)
    in
      (Env.singleSE(strid, S),
       OG.SINGLEsglstrbind(i', strid, out_psigexp, out_strexp))
    end

   (* Undefined structure bindings *)
 | IG.UNDEFsglstrbind(i, strid, psigexp) =>
    let
      val (Sigma, out_psigexp) = elab_psigexp(B, psigexp)
      val (N, S) = Stat.unSig(Sigma)
    in
      (Env.singleSE(strid, S),
       OG.UNDEFsglstrbind(okConv i, strid, out_psigexp))
    end

   (* Unguarded structure bindings *)
```

```
| IG.UNGUARDsglstrbind(i, strid, strexp) =>
   let
     val (S, out_strexp) = elab_strexp(B, strexp)
   in
     (Env.singleSE(strid, S),
      OG.UNGUARDsglstrbind(okConv i, strid, out_strexp))
   end
```

As can be seen, the result of elaboration of the single structure binding is a structure environment `Env.singleSE(strid, S)` obtained from the signature, rather than from the structure itself. Similarly, the result of elaboration of the unguarded structure binding is the structure environment obtained from its signature.

8.3 Functor bindings

Analogously as in the case of structures, the functors in EML behave like parameterized abstractions and the undefined functor bindings are introduced. Rules 99 and 99.1 describe this in detail:

$$\frac{\begin{array}{c} B \vdash \mathit{psigexp} \Rightarrow (N)S, \gamma_1 \qquad N \cap N \text{ of } B = \emptyset \qquad B' = B \oplus \{\mathit{strid} \mapsto S\} \\ B' \vdash \mathit{psigexp}' \Rightarrow \Sigma, \gamma_2 \qquad B' \vdash \mathit{strexp} \Rightarrow S', \gamma_3 \\ \Sigma \geq S'' \prec S' \qquad \langle B \vdash \mathit{funbind} \Rightarrow F, \gamma_4 \rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{psigexp}\ \texttt{)}\ \texttt{:}\ \mathit{psigexp}'\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{funbind} \rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, \Sigma)\}\ \langle + F \rangle,\ \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \langle \cdot \gamma_4 \rangle \end{array}} \tag{99}$$

$$\frac{\begin{array}{c} B \vdash \mathit{psigexp} \Rightarrow (N)S, \gamma_1 \qquad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{psigexp}' \Rightarrow \Sigma, \gamma_2 \\ N \cap N \text{ of } B = \emptyset \qquad \langle B \vdash \mathit{funbind} \Rightarrow F, \gamma_3 \rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{psigexp}\ \texttt{)}\ \texttt{:}\ \mathit{psigexp}'\ \texttt{= ?}\ \langle \texttt{and}\ \mathit{funbind} \rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, \Sigma)\}\ \langle + F \rangle,\ \gamma_1 \cdot \gamma_2 \langle \cdot \gamma_3 \rangle \end{array}} \tag{99.1}$$

The implementation of these rules was largely analogous to the implementation of the rules for structures.

The second premise in rule 99 is always satisfied in the EML Kit, because at the beginning of elaboration of *psigexp* the free names are chosen to be distinct from the names of the basis, and in the meantime no alpha-conversion of the result is made. As a result, we had no need of implementing any explicit check for this side condition.

9 The work related to the changes in presentation conventions

9.1 The type variable `num`

There is a special type variable `num` in the Definition of Extended ML. The definition of generalization is changed with respect to the Definition of Standard ML, and the definition of a substitution is added. In effect it is possible to capture formally the notion of *overloading resolution*.

In the SML definition, where the overloading resolution is described informally, it is only required to take place at each *topdec*. In the EML overloading resolution has to be done at *strdec* for declarations (see rule 57, where principality for *dec* is required), and at *axexp* for axioms (see Section 11.2).

Fortunately, the place where most SML compilers, including the ML Kit, choose to resolve overloading is *strdec*, so no general changes to the existing overloading resolution for declarations were necessary. The functions performing resolution of overloading inside declarations only had to be extended for the sake of the new EML constructs and put to work of resolving overloading inside axiom bodies.

9.2 The U sets

Let's suppose we have a guarding construct G (see [5], Section 4.6 for the definition of guarding constructs), which is a fragment of some larger piece of SML or EML code. The U set for G according to the SML definition is the set of explicit type variables scoped at G. The U set for G according to the EML definition is the set of explicit type variables occuring unguarded in G.

This is not the same, because the set of unguarded variables of a guarding construct G contains all the variables scoped at G and additionally every variable that is scoped at one of the outer guarding constructs, but occurs unguarded in G.

Let's look at the following example:

```
val d = (fn x : 'a => x, let val f = (fn y : 'a => y) in 1 end)
```

Here the SML U's are:	for the first val	— {`'a`},
	for the second	— $\emptyset$.
But the EML U's are:	for the first val	— {`'a`},
	for the second	— {`'a`}.

Fortunately, the function which computes the SML U's in the ML Kit, does it by first computing the EML U's and then subtracting variables scoped at outer guarding constructs. We extended the functions, computing EML U's, to deal with the EML-specific constructs and made the functions visible in the signatures of the modules containing them.

10 Implementation of the corrections to the SML definition

There are some known errors in the Definition of Standard ML (see [3] and [4]). The EML definition adapted the suggested corrections to these errors. This has obliged us to incorporate analogous changes to the EML Kit. We describe them in this section. See also Section B.1.2.

10.1 Local declarations

Rule 6 of the Static Semantics of EML looks as follows:

$$\frac{C \vdash \mathit{dec} \Rightarrow E, \gamma \qquad C \oplus E \vdash \mathit{exp} \Rightarrow \tau, U, \gamma' \qquad \text{tynames}\,\tau \subseteq T \text{ of } C}{C \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow \tau, U, \gamma \cdot \gamma'} \tag{6}$$

In rule 6 of the Static Semantics of SML there is no third premise. This may lead to an unsound elaboration (see [3] for details).

The third premise is implemented in the EML Kit, although the elaboration algorithm is sound even without this change, because of the side-effects used to generate fresh type names.

As a result of implementing this correction some SML programs, like

```
let
    datatype int_list =
        CONS of int * int_list
      | NIL
in
    CONS
end
```

are rejected by the EML Kit. This is signaled by a proper error message (see Section 14.3).

10.2 Match rules

In rule 16 the EML definition has $C \oplus VE$ instead of $C + VE$ found in the SML definition (see [4]):

$$\frac{C \vdash pat \Rightarrow (VE, \tau), U, \gamma \qquad C \oplus VE \vdash exp \Rightarrow \tau', U', \gamma'}{C \vdash pat \texttt{ => } exp \ \Rightarrow \tau \rightarrow \tau',\ U \cup U',\ \gamma \cdot \gamma'} \tag{16}$$

As in the previous case, the change is adopted in the EML Kit, although even without this change, the elaboration is sound, because of the side-effects used to generate fresh type names.

10.3 Well-formedness predicate

The well-formedness predicate, described in the EML definition, Section 5.3, is strengthened with respect to the SML definition well-formedness predicate. We have implemented the additional check of signature type structures' names and refrained from making a special check for functor signature type structures' names, because it's unnecessary in an implementation according to [7] and [1].

11 Proposed changes to the Static Semantics of EML

11.1 The side condition for rule 17

For compatibility reasons, a side condition similar to the one in rule 17 of the SML definition:

$$U \cap \mathrm{tyvars}\, VE' = \emptyset$$

should be added to rule 17 of the EML definition.

Recall the example from Section 9.2:

```
val d = (fn x : 'a => x, let val f = (fn y : 'a => y) in 1 end)
```

Here the SML U's are:	for the first val	— {'a},
	for the second	— $\emptyset$.
But the EML U's are:	for the first val	— {'a},
	for the second	— {'a}.

Now that we see the difference between SML U's and EML U's we would think that rule 17 would behave differently in SML than in EML. (Rule 17 is the only rule of the SML definition where the U's are used.)

Unexpectedly, if we drop the side condition in the SML rule 17, which makes the EML rule 17 and the SML rule 17 so modified identical up to notation (actually up to imperative features missing in

EML and traces not present in SML as well, but this is not important for this argument), we observe that both rules give identical results. This is because in rule 17 the U sets are used by adding them to the context. And this context always contains the explicit type variables scoped at the outer guarding constructs (the variables which make the difference between EML U's and SML U's) and so it doesn't matter if one adds unguarded variables or scoped variables.

Even more unexpectedly, if we do not remove the side condition from SML rule 17, but instead add the mentioned side condition to the EML rule 17, which again makes these rules almost identical, we get that the rules behave differently: the above example which type-checks in SML, doesn't type-check in EML with rule 17 enriched with the side condition.

The way out is to add to EML rule 17 not the SML side condition:

$$U \cap \mathrm{tyvars}\, VE' = \emptyset$$

but a modified one:

$$U \cap \mathrm{tyvars}\, VE' \subseteq U \text{ of } C$$

Now rule 17 of SML and rule 17 of EML behave identically.

To conform to the EML tradition of expressing properties using traces rather than other semantic objects, it's actually better to make this side condition a bit stronger:

$$U \cap \mathrm{tyvars}\, \mathrm{Clos}_C \gamma \subseteq U \text{ of } C$$

This modification allows us to add similar side conditions to rules 11.1–11.3, which describe EML-specific guarding constructs.

11.2 The principality requirement for axiom bodies

The second premise of rule 57.1 should be moved to rule 61.2, thus changing the place where the principality is imposed on axioms in structures (which doesn't do any harm) and adding the principality requirement for the axioms in signatures.

As a result specifications like:

```
sig
    axiom let
              fun g a b = a + b
          in
              true
          end
end
```

where overloading in axiom bodies cannot be resolved, become incorrect.

Moreover, the relation between the Modules part of the Static Semantics and the Core part is being made simpler and more unified, and a good reformulation of Trace is made possible (see Section 11.4).

11.3 Axiom bodies should be of type `bool`

In order to reject nonsensical axiom bodies, the requirement in rule 61.2 that $exp^\bullet$ has to be able to elaborate to `bool` should be changed to a stronger one, that whenever $exp^\bullet$ elaborates to τ, the τ is `bool`.

Programs like:

```
exception Oj
fun f a = raise Oj
axiom (f 1)
```

would be banned in this way. (The EML Kit issues an error message whenever this restriction is violated. See Section 14.3 for information about error messages.)

11.4 Reformulation of Modules Trace

The Core Trace is defined in a clean and precise way:

$$\begin{aligned}
&\mathrm{Trace} = \mathrm{Tree}(\mathrm{SimTrace} \uplus \mathrm{TraceScheme})\\
&\mathrm{SimTrace} = \mathrm{Type} \uplus \mathrm{Env} \uplus (\mathrm{Context} \times \mathrm{Type}) \uplus\\
&\qquad (\mathrm{Context} \times \mathrm{Env}) \uplus \mathrm{TyEnv} \uplus (\mathrm{VarEnv} \times \mathrm{TyRea})\\
&\mathrm{TraceScheme} = \uplus_{k \geq 0} \mathrm{TraceScheme}^{(k)}\\
&\mathrm{TraceScheme}^{(k)} = \mathrm{TyVar}^{k} \times \mathrm{Trace}
\end{aligned}$$

In the Version 1 of the Definition of Extended ML, Modules Trace is defined as follows:

$$\begin{aligned}
&\mathrm{Trace} = \mathrm{Tree}(\mathrm{SimTrace} \uplus \mathrm{TraceScheme} \uplus \mathrm{BoundTrace})\\
&\mathrm{BoundTrace} = \mathrm{NameSet} \times \mathrm{Trace}\\
&\mathrm{SimTrace} = \mathrm{SimTrace}_{\mathrm{COR}} \uplus \mathrm{StrName} \uplus \mathrm{Rea} \uplus \mathrm{VarEnv}
\end{aligned}$$

The TraceScheme mentioned here is the Core TraceScheme.

Why is this definition of Modules Trace not satisfactory?

- Core Trace, without any apparent reason, appear in the definition of Module Trace not as a whole $\mathrm{Trace}_{\mathrm{COR}}$, but as two separated fragments: TraceScheme and $\mathrm{SimTrace}_{\mathrm{COR}}$. This is obviously against principles of modular construction, and precludes thinking of the Core Trace as an abstract domain with certain operations, forcing one to remember the implementation details of Core Trace when studying Modules Trace.
- Rules 61.2, 61.1 and 57.1 of the Version 1 of the EML definition can be considered "type-correct", but only at the cost of either:
  - implicitly injecting $\mathrm{Trace}_{\mathrm{COR}}$ into Trace in rule 61.2 and implicitly extending the definition of Clos (which is defined in the EML definition to operate on the Core Trace and not Modules Trace) for use in rule 57.1, or
  - making the "type" of rule 61.2 '$B \vdash \mathit{axexp} \Rightarrow \gamma_{\mathrm{COR}}$' instead of '$B \vdash \mathit{axexp} \Rightarrow \gamma$', making the "type" of rule 61.1 '$B \vdash \mathit{ax} \Rightarrow \gamma_{\mathrm{COR}}$' (note that we thus assume that $\mathrm{Trace}_{\mathrm{COR}}$ is implemented as a Tree of some objects), and finally implicitly injecting TraceScheme into the Trace in rule 57.1, after using the Clos which has the type $\mathrm{Trace}_{\mathrm{COR}} \rightarrow \mathrm{TraceScheme}$ in this rule.
- Rule 57 doesn't seem to "type-check" in any way.

We propose the following definition of Modules Trace:

$$\begin{aligned}
&\mathrm{Trace} = \mathrm{Tree}(\mathrm{Trace}_{\mathrm{COR}} \uplus \mathrm{SimTrace} \uplus \mathrm{BoundTrace})\\
&\mathrm{BoundTrace} = \mathrm{NameSet} \times \mathrm{Trace}\\
&\mathrm{SimTrace} = \mathrm{StrName} \uplus \mathrm{Rea} \uplus \mathrm{VarEnv} \uplus \mathrm{TyEnv} \uplus \mathrm{Env}
\end{aligned}$$

Why is it better?

- The relation of Modules Trace to Core Trace is made clearer and more modular.
- When the place where Clos is performed is moved to rule 61.2, as described in Section 11.2, rules 57, 57.1, 61.1 and 61.2 "type-check" in a straightforward way.
- An implicit injection of $\mathrm{Trace}_{\mathrm{COR}}$ into Trace is involved when needed (and is not involved when is not needed, as e.g. when injecting TyEnv into Trace in rule 71, which of course doesn't have anything to do with Core).

Part IV
Miscellaneous

12 Abstract syntax tree

Bare language is a portion of SML *Full* language which suffices to express functionality of the whole SML[6] and on the other hand is small enough to present its syntax and semantics in a succinct way. The ML Kit defines several datatypes mimicking the structure of the grammar of SML *Bare* language. There is a distinct datatype corresponding to every syntactic class of Bare language. We call these datatypes *abstract syntax tree datatypes.*

Close correspondence between SML syntax and the ML Kit abstract syntax datatypes makes it straightforward to incorporate any changes and extensions to the former into the latter. Unfortunately, due to the fact that "Syntactically, SML is a nightmare" (see Section 3.4 of [1]), this correspondence breaks down in a few places. The differences are described in detail in Subsection 3.4.3 of [1], and concern issues like

- resolving of identifier status,
- resolving infixes in patterns, expressions and `fun`-bindings,

in a post-parsing pass.

EML language extends (and slightly modifies) the syntax of SML, and these changes are reflected in the abstract syntax tree datatypes of the EML Kit. Fortunately enough, the new syntax doesn't introduce any serious complications, so the changes we have made to the ML Kit are straightforward and faithfully correspond to changes in the syntax. Here we give the additions to the ML Kit abstract syntax datatypes, which were needed to introduce the new syntactic constructs of the *Core* language part of EML.

```
datatype atexp =    ...
      UNDEFatexp of info               (* undefined value *)

and exp = ...
      COMPARexp of info * exp * exp |   (* comparison *)
      EXIST_QUANTexp of info * match |  (* existential quantifier *)
      UNIV_QUANTexp of info * match |   (* universal quantifier *)
      CONVERexp of info * exp |         (* convergence predicate *)

and dec =  ...
      EQTYPEdec of info * typbind |     (* equality type declaration *)

and typbind = ...
      QUEST_TYPBIND of info * tyvar list * tycon * typbind Option
                                        (* question mark type binding *)
```

Changes incurred to the abstract syntax tree datatypes for the *Modules* language include the following:

[6] This is achieved through so called derived forms, which were treated informally in [8], and were given a formal definition in Appendix B of [5], in the case of EML.

```
datatype strdec = ...
  AXIOMstrdec of info * ax |          (* axiom *)

and ax =
  AXIOMax of info * axexp * ax Option  (* axiom *)

and axexp =
  AXIOM_EXPaxexp of info * exp        (* axiomatic expression *)

and strbind =
  STRBIND of info * sglstrbind * strbind Option
                                      (* structure binding *)

and sglstrbind =
  SINGLEsglstrbind of info * strid * psigexp * strexp |
                                      (* single structure binding *)
  UNDEFsglstrbind of info * strid * psigexp |
                                      (* undefined structure binding *)
  UNGUARDsglstrbind of info * strid * strexp
                                      (* unguarded structure bindgin *)

and psigexp =
  PRINCIPpsigexp of info * sigexp     (* principal signature *)

and sigbind =
  SIGBIND of info * sigid * psigexp * sigbind Option

and spec = ...
  AXIOMspec of info * axdesc |        (* axiom *)

and axdesc =
  AXDESC of info * specexp * axdesc Option

and specexp =
  SPECEXP of info * strdec * axexp

and funbind =
  FUNBINDfunbind of info * funid * strid * psigexp * psigexp * strexp * funbind Option |
                                      (* functor binding *)
  UNDEFfunbind of info * funid * strid * psigexp * psigexp * funbind Option
                                      (* undefined functor binding *)
```

The dynamic semantics of both SML and EML define the so called *reduced syntax* (see Sections 6.1 and 7.1 of [8] and [5] for details), to reflect the fact, that certain syntactic constructs are irrelevant to evaluation of ML programs (they involve types, axioms, sharing equations and the like, which are exploited in the elaboration). The ML Kit however (and the EML Kit keeps this unchanged) does not bother with explicitly transforming abstract syntax representation of programs into the reduced syntax. The price one pays for this is an obligation to write some code in the (E)ML Kit evaluator, which handles cases of superfluous syntax. As it has no counterpart in the rules of dynamic semantics, the result of evaluation of such code should have the same effect as simply ignoring it.

13 Evaluation

13.1 Core evaluation

The EML Dynamic Semantics for the Core is very similar to the SML Dynamic Semantics for the Core. The only major additions are the rules describing the evaluation of the new Core language constructs. They are listed in Table 11.

Table 11: **The rules describing the evaluation of the new Core language constructs**

rule number	*form of the construct*	*name of the construct*	*phrase class*
109.1	`?`	undefined value	*atexp*
118.1	$exp_1^\bullet$ `==` $exp_2^\bullet$	comparison	*exp*
118.2	`exists` $match^\bullet$	existential quantifier	*exp*
118.3	`forall` $match^\bullet$	universal quantifier	*exp*
118.4	$exp^\bullet$ `terminates`	convergence predicate	*exp*

Let's look closer at rule 118.4:

$$\frac{}{s, E \vdash exp^\bullet\ \texttt{terminates} \Rightarrow [\texttt{NoCode}], s_\perp} \tag{118.4}$$

It says that $exp^\bullet$ `terminates` evaluates to the package [`NoCode`], whatever the environment E and the expression $exp^\bullet$ are (they are only assumed to be well-formed and type-correct).

And now the piece of the EML Kit code corresponding to this rule:

```
fun evalExp(E, exp) =
  case exp of
  ...
    | CONVERexp(_, _) => raiseNoCode()
    ...
```

`CONVERexp` is the name of the abstract syntax tree node (see Section 12) representing the `terminates` construct. `raiseNoCode` is a function implementing the EML-specific special exception `NoCode` (see Section B.1.1).

All the rules of the Dynamic Semantics are deterministic, in the sense that for every given phrase and context there is at most one derivation of a value for the phrase in this context. This justifies treating the Dynamic Semantics as a deterministic recipe for evaluating EML phrases, and allows the straightforward implementation of the rules, those mentioned in Table 11 in particular.

13.2 Modules evaluation

There are some minor changes to dynamic semantics of SML concerning rules 169, 171.1 and 175, which have been implemented. More significant changes and additions are gathered in table 12. Among them rules 164.1, 169.1 and 169.3 have been implemented in the EML Kit. Rules 169.2 and 187.1 are handled, but do not behave correctly, because functions `TrivEnv` and `TrivStrExp` described in Section 7.2 of [5] have not been implemented yet. Finally rules 176.1, 184.1 and 184.2 are still not handled at all.

Table 12: **New rules describing the evaluation of Modules language constructs**

rule number	*form of the construct*	*name of the construct*
164.1	`axiom` *ax*	axiom
169.1	*strid* : *psigexp* = *strexp*	single structure binding
169.2	*strid* : *psigexp* = `?`	single structure binding
169.3	*strid* = *strexp*	single structure binding
176.1	`datatype` *datdesc*	datatype
184.1	*tyvarseq tycon* = *condesc* ⟨ `and` *datdesc* ⟩	datatype description
184.2	*con* ⟨ | *condesc* ⟩	constructor description
187	*funid* (*strid* : *psigexp*) : *psigexp*′ = *strexp*	functor binding
187.1	*funid* (*strid* : *psigexp*) : *psigexp*′ = `?`	undefined functor binding

14 Output

14.1 Pretty-printing

The ML Kit is not merely an interpreter of SML code. It is an interactive system, which can support the user with various information concerning lexical analysis, parsing, elaboration and evaluation of SML programs which are fed to it. In order to be able to present such data to the user in a neat and tidy way, several pretty-printing routines are needed. The ML Kit designers have written a simple, but quite general and powerful enough pretty-printing facility, which can print everything one can represent as a "tree of strings". Here is definition of the `StringTree` datatype, elements of which can be printed by `PrettyPrint` module either in a single line, or as a list of lines on a page of given width:

```
signature PRETTYPRINT =
  sig
    datatype StringTree = LEAF of string
                        | NODE of {start : string, finish: string, indent: int,
                                   children: StringTree list,
                                   childsep: childsep}
    and childsep = NONE | LEFT of string | RIGHT of string
    ...
  end
```

An element of `StringTree` datatype is either a `LEAF`, consisting of a string which will not be split over lines of output; or it is a `NODE` which corresponds to a text beginning with a "start" string, ending with a "finish" string, with a list of "children" coming in between, separated with "child separator" in case there are two or more of them.

For example to pretty-print SML programs represented in the Kit as elements of abstract syntax tree datatypes, it is enough to write functions which transform them to the `StringTree` datatype, and then just use general pretty-printing routine supported by `PrettyPrint` module. Thus in order to enable pretty-printing of new syntactic constructs of EML language, we added to the ML Kit the following:

- a few functions corresponding to new syntactic classes of EML (see Section 12); for example function `layoutAx` is used to make a `StringTree` out of an abstract syntax tree for an *axiom*:

```
fun layoutAx ax : StringTree =
  let
    val axs = makeList (fn AXIOMax(_, _, opt) => opt) ax

    fun layout1(AXIOMax(_, axexp, _)) =
      PP.NODE{start="", finish="", indent=0,
              children=[layoutAxexp axexp],
              childsep=PP.NONE
             }
  in
    PP.NODE{start="", finish="", indent=0,
            children=map layout1 axs,
            childsep=PP.LEFT " and "
           }
  end
```

- a bunch of matches in existing functions, corresponding to new syntactic constructs introduced to existing syntactic classes of SML; for example due to introduction of *existential quantifiers* to the syntax of expressions, we included the following match to function `layoutExp`:

```
fun layoutExp exp : StringTree =
  case exp
  ...
     | EXIST_QUANTexp(_, match) =>
         PP.NODE{start="exists ", finish="", indent=7,
                 children=[layoutMatch match],
                 childsep=PP.NONE

                }
```

14.2 Error detection and reporting

The ML Kit handles error detection and reporting in a clean and uniform way. Instead of eagerly printing messy error messages on the screen every time something goes wrong, it collects all such data into objects called *reports*. These reports are then accumulated at the top level of the system, where they are finally processed and printed out in an appropriate format.

Moreover, some of the report generation is being deferred during elaboration, in order to simplify the structure of the elaborator. Any error information detected during elaboration is placed instead into error info nodes in the abstract syntax. Thus after elaboration one has to browse thorough abstract syntax to collect this data if there is any. This error info collection is done by a family of functions implemented in module `ErrorTraverse`. For example, in order to be able to extract error info from an abstract syntax tree of an *axiom*, we added the following straightforward definition of function `walk_Ax` to the `ErrorTraverse` functor:

```
fun walk_Ax ax =
  case ax
    of AXIOMax(i, axexp, ax_opt) =>

         check i // walk_Axexp axexp // walk_opt walk_Ax ax_opt
```

14.3 The EML Kit error messages

Error info collected during elaboration is then converted to a *report* (which is just a printable text divided into lines of output) by means of `reportInfo` function implemented by `ErrorInfo` module. In order to report new types of errors which might occur after adding new features to ML language, one has to add

- new constructors to the `info` datatype,
- corresponding matches to `reportInfo` functions to handle these new constructors.

Work on the implementation of EML static analysis described in part III led us to introduce the following error message constructors to the `info` datatype:

```
datatype info =
 (* Core errors: *)
 ...
  | SHOULD_ADMIT_EQ of TypeFcn list
  | LOCAL_TYNAMES
 (* General module errors: *)
 ...
  | AXEXP_SHOULD_BE_BOOL
  | UNGUARD_EXPLICIT_TV_IN_AXEXP
```

These constructors are planted into info nodes of abstract syntax in cases when respective errors come out during static analysis. Then after elaboration of erroneous phrases the following error messages are issued by the system (as mentioned above, report generation is done by `reportInfo` function of `ErrorInfo` module):

- `SHOULD_ADMIT_EQ(tfnl)`:
  `Type function(s): <tfnl> should admit equality`
- `LOCAL_TYNAMES`:
  `Local tyname(s) escape from a let expression`
- `AXEXP_SHOULD_BE_BOOL`:
  `Axiomatic expressions must be of type bool`
- `UNGUARD_EXPLICIT_TV_IN_AXEXP`:
  `Unguarded explicit type variable(s) in axiomatic expression`

Part V
Appendixes

A Release Notes

We welcome any bug-reports and comments; write to `eml@zls.mimuw.edu.pl`.

A.1 Copyright notice

A.2 Getting the EML Kit

The EML Kit binaries are available through anonymous ftp from `zls.mimuw.edu.pl`. You should `cd` to the `/pub/mikon/` directory and get the `eml1.tgz` archive.

It contains:

- `README`,
- `ANNOUNCEMENT` — some information about this release,
- `emlkit` — the Sun SPARC executable
- `emlkit2eml`, `emlkit2eml_types_only` — shell scripts
- `emlkit2eml.cmd`, `emlkit2eml_types_only.cmd` — SML scripts
- `COPYING` — text of the GNU General Public License

A.3 Installation

After ungziping and untaring the `eml1.tgz` archive execute the script `emlkit2eml`. It will take a while... Then execute `emlkit2eml_types_only` and after some time you should have in your directory two additional files: `eml` and `eml_types_only`. The `eml` file is what you would probably use the most. Place it in a proper directory.

The file `emlkit` is an EML Kit executable with the ML Kit's style user interface. The `eml` is just the exported `eval()` and `eml_types_only` is the exported `elab()`. Unlike the ML Kit all of these three executables have `use` visible at the top-level.

The executable `eml` parses, type-checks and evaluates EML programs. Its user interface is similar to the one of the SML/NJ compiler.

The executable `eml_types_only` parses and type-checks only. Although `use` is visible in it at the top-level, it won't import files, because it doesn't evaluate.

B The State of the System

The EML Kit is being used with success by many people in Warsaw and in Edinburgh, but being a large and complex software system, based on formidable, but changing and not error-immune foundations, it suffers from small but persistent deficiencies. In this chapter we describe known bugs and not fully functional features, found in the current version of the EML Kit.

B.1 Problems inherited from the ML Kit

B.1.1 Special exceptions

The special exceptions of the SML definition, in particular `Match` and `Bind`, are not implemented in the ML Kit. As in the EML Dynamic Semantics for the Core new special exceptions are defined and heavily used, we are prevented from giving our implementation of the EML Dynamic Semantics for the Core its full functionality (see Section 13.1).

In the event someone heavily uses EML specific constructions outside axioms (e.g. for experimental purposes), evaluation mechanisms of the EML Kit may unexpectedly fail with:

```
Unimplemented: raiseNoCode
System Crash: Reentering...
```

or a similar message. Fortunately the executable `eml_types_only` provided with this EML Kit release allows one to study the type behavior of a program without invoking the evaluation phase. Similar effects can be obtained by invoking `elab()`, or `elabFile` functions from the `emlkit` executable .

B.1.2 Errors in the SML definition

There are errors in the SML definition (see [3] and [4] for details). Some of them are "implemented" in the ML Kit. We have corrected most of these (see Section 10). The most notable of those that remained is the error concerning Exception Environments and the one about identifier status (see [3] pp. 24 and 21).

Fortunately the ML Kit performs an unsound elaboration caused by the identifier status bug only in very pathological cases, and seems to avoid troubles with Exception Environment at all, thanks to changes in the semantics of signature matching.

B.1.3 Nonfunctional features of the ML Kit

There are some bugs and unimplemented or nonfunctional features in the ML Kit. Some of these occur in most other SML compilers as for example problems with parsing ambiguity, some do not. The EML Kit inherits most of these problems. None of them is very dangerous, but some decrease the comfort of the EML Kit use, as for example the fact that the interpreter doesn't check if patterns and bindings are exhaustive, or sometimes doesn't print position information when reporting errors.

B.2 Other problems

B.2.1 Trace

The definition of Trace has been changing rapidly lately, and as for now the version of the Definition of Extended ML which will contain the final definition of Trace is not finished. As a result Trace is not yet implemented in the EML Kit.

Most of the effects of this omission are very esoteric and probably none of the EML Kit users will ever stumble across one. There is one exception though. The axioms in signatures are not elaborated in the equality-principal signatures, as they should be. The effects of this bug are of two kinds.

First, if one uses ordinary equality ("=") in axioms in a signature, the EML Kit may reject an axiom even if it's correct. This happens because some of the equality types in the signature are not known to be equality types before the signature is "equality-principalized". A way around is to use double equality ("==") in axioms, which is by the way considered by some an element of the EML good programming style.

Second, if one uses many sharing equations and axioms in the same signature, some of the axiom bodies may be wrongly considered incorrect by the EML Kit. Reasons are similar as for the problem with equality. No one has ever come upon this bug, yet.

References

[1] L. Birkedal, N. Rothwell, M. Tofte and D. N. Turner. *The ML Kit, Version 1* (1993).

[2] L. Damas and R. Milner. Principal type schemes for functional programs. *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212. ACM, New York (1982).

[3] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Report ECS-LFCS-93-257, Univ. of Edinburgh (1993).

[4] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML – Addenda. Univ. of Edinburgh (1995).

[5] S. Kahrs, D. Sannella and A. Tarlecki. *The Definition of Extended ML.* Report ECS-LFCS-94-300, Univ. of Edinburgh (1994).

[6] S. Kahrs and D. Sannella and A. Tarlecki. The semantics of Extended ML: a gentle introduction. *Proc. Workshop on Semantics of Specification Languages*, Utrecht, 1993. Springer Workshops in Computing, 186–215 (1994).

[7] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press (1991).

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

[9] D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park, 1990. Springer Workshops in Computing, 99–130 (1991).

[10] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[11] D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).