LFCS

**Laboratory for Foundations of Computer Science**
Department of Computer Science - University of Edinburgh

# Syntax, Typechecking and Dynamic Semantics for Extended ML

by

Donald Sannella
Fabio da Silva

Syntax, Typechecking and Dynamic Semantics ..........

**LFCS Report Series**

**ECS-LFCS-89-101**
**(also published as CSR-320-89)**

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

**December 1989**

# Syntax, Typechecking and Dynamic Semantics for Extended ML

Donald Sannella Fabio da Silva

**Abstract**

This document defines the syntax, typechecking and dynamic semantics of the Extended ML specification/programming language. The reader is assumed to be familiar with the notation used in the formal definition of Standard ML.

## 1 Introduction

Extended ML (EML) is a framework for the formal development of programs in the Standard ML (SML) programming language from high-level specifications of their required input/output behaviour. The EML language is a "wide-spectrum" language which encompasses both specifications and executable programs in a single unified framework. It is a simple extension of SML in which axioms are permitted in module interfaces and in place of code in module bodies. This allows all stages in the development of a program to be expressed in the EML language, from the initial high-level specification to the final program itself and including intermediate stages in which specification and program are intermingled.

This document defines the syntax, static semantics and dynamic semantics of the EML language as an extension and modification of SML. The reader is assumed to be acquainted with the formal definition of SML [HMT89]. The present document has a similar structure: Section 2 covers the syntax of the core and module languages, Section 3 gives static semantics (typechecking) rules, Section 4 covers dynamic semantics, and the Appendices cover derived forms. Everything which is not explicitly mentioned here (e.g. lexical matters) remains as in SML. Section 5 contains some remarks on the rationale for the definition in this document.

The motivation, theoretical underpinnings and use of EML are discussed in [ST85], [ST86], [ST89] and [San90]. The interested reader should consult these for more information. The present document is intended as a definition of the EML language for use by implementors of the language. One important omission is a definition of EML signature matching which takes axioms into account. Since this involves arbitrarily complex theorem proving and is therefore undecidable, it is regarded as a separate problem to be addressed by an EML program development environment. The static semantics of EML as given here treats only those aspects of signature matching which can be handled by a compiler. This means that axioms are treated as formal comments which are parsed and typechecked but have no other effect.

## 2 Syntax

### 2.1 Syntax of the Core

The core syntax of EML is a proper extension of the core syntax of SML. New reserved words are `axiom`, `forall`, `exists` and `?`. We introduce six new phrase classes in Figure 1, and use the variable *logatexp* to range over LogAtExp, etc.

| | |
|---|---|
| LogAtExp | logical atomic expressions |
| LogExp | logical expressions |
| LogMatch | logical matches |
| LogMrule | logical match rules |
| LogExpRow | logical expression rows |
| Ax | axioms |

Figure 1: Core Phrase Classes for EML

LogExp (and related syntactic classes) are introduced in order to simplify the typechecking of axiom declarations. This is better explained in Section 3.1.

The new grammar rules for the EML core language are given in Figure 2. As in the SML definition, we first give the syntax of the *bare* language. The derived forms of LogExp, and their equivalents in the bare language, are given in Appendix A. The full grammar for LogExp is given in Appendix B. We exclude function abstraction from the syntax of LogExp; therefore conditional expressions must be included in the bare syntax of LogExp. The new syntax for declarations, atomic expressions and type expressions are additions to those for SML. The grammar for those syntactic classes that do not appear in Figure 2 remains as in the definition of SML.

### 2.2 Syntax of Modules

The grammar for modules in EML introduces minor alterations in that for SML. In this sense, the grammar of EML is not a proper extension of the grammar of SML. The grammar for the syntactic classes that do not appear below remains as in the definition of SML. Functor and structure bindings are required to include explicit output signatures, but may have undefined bodies (via the use of ?). The syntax of Spec is extended to include axioms. The relevant rules in the grammar of modules are given in Figure 3.

## 3 Static Semantics

### 3.1 Static Semantics for the Core

The semantic objects are the same as in the SML definition. We give below the rules for typing the undefined value ?, quantified and conditional expressions, axiom declarations, equality type declarations and the undefined type ?. We do not give rules for LogMatch, LogMrule, LogExpRow, LogAtExp, and for most of the forms of LogExp, which are handled the same as the similar forms in SML. The rules for the derived forms of LogExp are also omitted. Apart from the following rules, typechecking in EML remains as in SML.

| | | | |
|---|---|---|---|
| *logatexp* | ::= | *scon* | special constant |
| | | ⟨op⟩*longvar* | value variable |
| | | ⟨op⟩*longcon* | value constructor |
| | | `{` ⟨*logexprow*⟩ `}` | record |
| | | `let` *dec* `in` *logexp* `end` | local declaration |
| | | `(` *logexp* `)` | |
| *logexprow* | ::= | *lab* `=` *logexp* ⟨ `,` *logexprow*⟩ | logical expression row |
| *logexp* | ::= | *logatexp* | logical atomic |
| | | *logexp logatexp* | application (L) |
| | | $logexp_1$ *id* $logexp_2$ | infixed application |
| | | *logexp* `:` *ty* | typed (L) |
| | | `exists` *logmatch* | existential quantifier |
| | | `forall` *logmatch* | universal quantifier |
| | | `if` $logexp_1$ `then` $logexp_2$ `else` $logexp_3$ | conditional |
| *logmatch* | ::= | *logmrule* ⟨`\|` *logmatch*⟩ | |
| *logmrule* | ::= | *pat* `=>` *logexp* | |
| *atexp* | ::= | ? | undefined value |
| | | ⋮ | |
| *dec* | ::= | `axiom` *ax* | axiom |
| | | `eqtype` *typbind* | equality type declaration |
| | | ⋮ | |
| *ax* | ::= | *logexp* ⟨and *ax*⟩ | axioms |
| *ty* | ::= | ? | undefined type |
| | | ⋮ | |

Figure 2: Core Grammar for EML

| | | | |
|---|---|---|---|
| *strbind* | ::= | *strid* `:` *sigexp* `=` *strexp* ⟨and *strbind*⟩ | structure binding |
| | | *strid* `:` *sigexp* `=` ? ⟨and *strbind*⟩ | undefined structure binding |
| *funbind* | ::= | *funid* `(` *strid* `:` *sigexp* `)` : *sigexp*′ = *strexp* ⟨and *funbind*⟩ | functor binding |
| | | *funid* `(` *strid* `:` *sigexp* `)` : *sigexp*′ = ? ⟨and *funbind*⟩ | undefined functor binding |
| *spec* | ::= | `axiom` *ax* | axiom |
| | | ⋮ | |

Figure 3: Modules Grammar for EML

The introduction of the phrase class LogExp simplifies the typechecking of axioms and quantified expressions, for it is then possible to isolate the occurrences of quantification and equality over non-equality types to axiom declarations by syntactic means. As can be seen in rule (7), axioms are typechecked in a context in which the equality function can be applied to values of any type.

## Atomic Expressions

$\boxed{C \vdash \mathit{atexp} \Rightarrow \tau}$

$$\frac{\texttt{'a} \text{ fresh in } C}{C \vdash \texttt{?} \Rightarrow \texttt{'a}} \tag{1}$$

*Comments:*

(1) An undefined value receives type `'a`, i.e. the most general type. Therefore, the type of the undefined value unifies with any type, allowing the undefined value to appear in any expression.

## Logical Expressions

$\boxed{C \vdash \mathit{logexp} \Rightarrow \tau}$

$$\frac{C \vdash \mathit{logmatch} \Rightarrow \tau \to \texttt{bool} \qquad \tau \notin \text{FunType}}{C \vdash \texttt{exists}\ \mathit{logmatch} \Rightarrow \texttt{bool}} \tag{2}$$

$$\frac{C \vdash \mathit{logmatch} \Rightarrow \tau \to \texttt{bool} \qquad \tau \notin \text{FunType}}{C \vdash \texttt{forall}\ \mathit{logmatch} \Rightarrow \texttt{bool}} \tag{3}$$

$$\frac{C \vdash \mathit{logexp}_1 \Rightarrow \texttt{bool} \qquad C \vdash \mathit{logexp}_2 \Rightarrow \tau \qquad C \vdash \mathit{logexp}_3 \Rightarrow \tau}{C \vdash \texttt{if}\ \mathit{logexp}_1\ \texttt{then}\ \mathit{logexp}_2\ \texttt{else}\ \mathit{logexp}_3 \Rightarrow \tau} \tag{4}$$

*Comments:*

(2),(3) The reason for the constraint on $\tau$ is to restrict quantification to be first-order. The constraint does not quite achieve this result, since the following type is not in FunType:

```
datatype foo = mkfoo of int -> int
```

but such type declarations will not appear in programs developed using EML anyway.

(4) We give the static semantics of conditional expressions since they are now part of the bare syntax of LogExp. Nevertheless, conditional expressions are still derived forms of Exp.

## Declarations

$\boxed{C \vdash \mathit{dec} \Rightarrow E}$

$$\frac{C \vdash \mathit{ax} \Rightarrow \{\}}{C \vdash \texttt{axiom}\ \mathit{ax} \Rightarrow \{\} \text{ in Env}} \tag{5}$$

$$\frac{C \text{ of } B \vdash \mathit{typbind} \Rightarrow \mathit{TE} \qquad \forall(\theta, \mathit{CE}) \in \text{Ran}\,\mathit{TE},\ \theta \text{ admits equality}}{B \vdash \texttt{eqtype}\ \mathit{typbind} \Rightarrow \mathit{TE} \text{ in Env}} \tag{6}$$

**Axioms** $\boxed{C \vdash ax \Rightarrow E}$

$$\frac{\begin{array}{c} C' = C \setminus \{\texttt{ref} \mapsto \forall\, \texttt{'\_a}\,.\, \texttt{'\_a} \rightarrow \texttt{'\_a ref},\ \texttt{:=} \mapsto \forall \texttt{'a}.\ \texttt{'a ref} * \texttt{'a} \rightarrow \texttt{unit}\} \\ C' + \{\, = \mapsto \forall\texttt{'a}.\ \texttt{'a} * \texttt{'a} \rightarrow \texttt{bool}\} \vdash \mathit{logexp} \Rightarrow \texttt{bool} \qquad \langle C \vdash ax \Rightarrow \{\}\rangle \end{array}}{C \vdash \mathit{logexp}\ \langle \texttt{and}\ ax\rangle \Rightarrow \{\}} \tag{7}$$

*Comments:*

(7) Axioms are elaborated in the context C altered in two ways. First, `ref` and `:=` values are taken out of C yielding $C'$. Then, $C'$ is extended with a new type for the equality function, in order to allow equality between values of every type (even types which do not admit equality). This does not cause problems because `axiom` declarations are not executable.

**Type Expressions** $\boxed{C \vdash ty \Rightarrow \tau}$

$$\frac{t \notin (T \text{ of } C)}{C \vdash\ ?\Rightarrow\ t} \tag{8}$$

*Comments:*

(8) A question mark type gets a new type name every time it is elaborated. This prevents the question mark type from unifying with any other type. Note that the ? type is not the type of the ? value.

## 3.2 Static Semantics for Modules

The semantic objects are the same as in the static semantics for SML modules. Signature matching in EML functor and structure declarations is more strict than in SML. In SML, the output signature of a structure/functor is taken to be the explicit output signature augmented by any information about type and structure sharing which can be inferred from the body (sharing *by construction* in [Mac86]). In EML, structures are like *abstractions* [Mac86] and functors are like parameterised abstractions rather than parameterised structures: the output signature of a structure/functor is taken to be exactly the explicit output signature with no added sharing.

In order to achieve this we need to introduce a new semantic function:

$$\psi\ :\ \text{Sig} \rightarrow \text{Str}$$

defined as follows:

$$\psi(N(S))\ =\ S'$$

where:

name of S = name of $S'$, and
Env of $S'$ is obtained from Env of S by renaming all bound names of S in the following way:

if $m \in N$ and $m'$ is a fresh name in B (type name or structure name), then replace all occurrences of $m$ in S for $m'$.

The inference rules are:

**Structure Binding** $\boxed{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}}$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow S \qquad B \vdash \mathit{sigexp} \Rightarrow \Sigma\ ,\ \Sigma \geq S' \prec S\ ,\ S'' = \psi(\Sigma) \\ \langle B + \text{names}\, S'' \vdash \mathit{strbind} \Rightarrow \mathit{SE} \rangle \end{array}}{B \vdash \mathit{strid}\ :\ \mathit{sigexp} = \mathit{strexp}\ \langle \text{and } \mathit{strbind} \rangle \Rightarrow \{\mathit{strid} \mapsto S''\}\ \langle + \ \mathit{SE} \rangle} \tag{9}$$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow \Sigma\ ,\ S = \psi(\Sigma) \\ \langle B + \text{names}\, S \vdash \mathit{strbind} \Rightarrow \mathit{SE} \rangle \end{array}}{B \vdash \mathit{strid}\ :\ \mathit{sigexp} = \ ?\ \langle \text{and } \mathit{strbind} \rangle \Rightarrow \{\mathit{strid} \mapsto S\}\ \langle + \ \mathit{SE} \rangle} \tag{10}$$

**Functor Bindings** $\boxed{B \vdash \mathit{funbind} \Rightarrow F}$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (N)S \qquad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{strexp} \Rightarrow S' \\ B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow \Sigma',\ \Sigma' \geq S'' \prec S' \\ S''' = \psi(\Sigma') \qquad N''' = \text{names}\, S''' \setminus ((N \text{ of } B) \cup N) \\ \langle B \vdash \mathit{funbind} \Rightarrow F \rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ (\ \mathit{strid}\ :\ \mathit{sigexp}\ )\ :\ \mathit{sigexp}' = \mathit{strexp}\ \langle \text{and } \mathit{funbind} \rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, (N''')S''')\}\ \langle + \ F \rangle \end{array}} \tag{11}$$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (N)S \qquad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow \Sigma' \\ S'' = \psi(\Sigma') \qquad N'' = \text{names}\, S'' \setminus ((N \text{ of } B) \cup N) \\ \langle B \vdash \mathit{funbind} \Rightarrow F \rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ (\ \mathit{strid}\ :\ \mathit{sigexp}\ )\ :\ \mathit{sigexp}' = \ ?\ \langle \text{and } \mathit{funbind} \rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, (N'')S'')\}\ \langle + \ F \rangle \end{array}} \tag{12}$$

**Specifications** $\boxed{B \vdash \mathit{spec} \Rightarrow E}$

$$\frac{C \vdash \mathit{ax} \Rightarrow \{\}}{C \vdash \texttt{axiom}\ \mathit{ax} \Rightarrow \{\} \text{ in Env}} \tag{13}$$

## 4 Dynamic Semantics

### 4.1 Dynamic Semantics for the Core

The semantic objects are the same as in the core dynamic semantics of SML. Apart from the new constructs, EML's dynamic semantics is exactly the same as for SML. A new basic value `Incomplete` is introduced, which is the semantic value of `?`. The new exception `NoCode` is also introduced. The new exception `NoCode` is not added to the set of basic exceptions; rather it is handled in the rule for value binding (rule (16)). This prevents the exception `NoCode` from being caught by an explicit handler in the program. The rules are:

**Atomic Expressions** $\boxed{E \vdash \mathit{atexp} \Rightarrow v/p}$

$$\frac{}{E \vdash\ ?\ \Rightarrow [\texttt{NoCode}]} \tag{14}$$

$$\frac{E(\mathit{longvar}) = \texttt{Incomplete}}{E \vdash \mathit{longvar} \Rightarrow [\texttt{NoCode}]} \tag{15}$$

*Comments:*

(14) The evaluation of ? always raises an exception. This exception is caught before any binding takes place (rule (16)).

(15) When a variable's value is `Incomplete`, the variable always evaluates to an exception, indicating that no code exists for that binding.

### Value Bindings

$$\boxed{E \vdash \mathit{valbind} \Rightarrow VE/p}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow [\texttt{NoCode}] \qquad E, \texttt{Incomplete} \vdash \mathit{pat} \Rightarrow VE \qquad \langle E \vdash \mathit{valbind} \Rightarrow VE' \rangle}{E \vdash \mathit{pat} \texttt{ = } \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle \Rightarrow VE\ \langle + \ VE' \rangle} \tag{16}$$

*Comments:*

(16) If the exception `NoCode` is raised while evaluating the expression, the exception is caught before doing the binding. Then the binding is done with the value `Incomplete`. In this rule, the *exception convention* of Section 6.7 in [HMT89] is not obeyed.

## 4.2 Dynamic Semantics for Modules

The semantic objects are the same as in the dynamic semantics of SML modules. We give the dynamic semantics for structure and functor bindings when the body of the structure or functor is undefined (i.e. ?). The dynamic semantics for normal functors and structures are handled as in SML. To generate a dynamic environment from an interface we define the operation

$$\epsilon \ : \ \mathrm{Int} \ \rightarrow \ \mathrm{Env}$$

as follows:

$$\epsilon(I) \ = \ (SE, VE, EE)$$

where:

$$\begin{aligned} VE &= \{\forall id \in \mathit{vars}\,\mathrm{of}\, I : id \mapsto \texttt{Incomplete}\} \\ EE &= \{\forall id \in \mathit{excons}\,\mathrm{of}\, I : id \mapsto id\} \\ SE &= \{\forall id \in \mathrm{Dom}\, IE\ \mathrm{of}\, I : id \mapsto \epsilon(IE(id))\} \end{aligned}$$

We also define:

$$\Gamma \ : \ \mathit{Env} \ \rightarrow \ \mathit{StrExp}$$

to be a function that generates a structure expression from an environment. This is necessary because a functor closure requires a structure expression as a component. The definition of Γ is:

$$\Gamma(E) \ = \ \texttt{struct}\ \mathit{strdec}\ \texttt{end}$$

where *strdec* is:

$$\begin{array}{l}\forall id \in \mathrm{Dom}\, VE \text{ of } E : \texttt{val}\ id = ?\\ \forall id \in \mathrm{Dom}\, EE \text{ of } E : \texttt{exception}\ id\\ \forall id \in \mathrm{Dom}\, SE \text{ of } E : \texttt{structure}\ id = \Gamma(SE(id))\end{array}$$

The semantic rules are:

**Structure Bindings** $\boxed{B \vdash strbind \Rightarrow SE/p}$

$$\frac{\mathrm{Inter}\, B \vdash sigexp \Rightarrow I \qquad \langle B \vdash strbind \Rightarrow SE\rangle}{B \vdash strid\ :\ sigexp = \ ?\ \langle \text{and } strbind\rangle \Rightarrow \{strid \mapsto \epsilon(I)\}\ \langle +\ SE\rangle} \quad (17)$$

**Functor Bindings** $\boxed{B \vdash funbind \Rightarrow F}$

$$\frac{\begin{array}{c}\mathrm{Inter}\, B \vdash sigexp \Rightarrow I \qquad \mathrm{Inter}\, B + \{strid \mapsto I\} \vdash sigexp' \Rightarrow I'\\ E = \epsilon(I') \qquad \langle B \vdash funbind \Rightarrow F\rangle\end{array}}{\begin{array}{l}B \vdash funid\ (\ strid\ :\ sigexp\ )\ :\ sigexp' = \ ?\ \langle \text{and } funbind\rangle \Rightarrow\\ \qquad \{funid \mapsto (strid : I, \Gamma(E) : I', B)\}\ \langle +\ F\rangle\end{array}} \quad (18)$$

## 5 Rationale

The definition given here adheres in most respects to the usage of EML in [ST89] and [San90]. As in those papers, the notation of first-order equational logic is used for axioms. The syntax for `forall` and `exists` is by analogy with the syntax for `fn`. Axioms are *not* implicitly universally quantified over all their free variables; the additional redundancy introduced by requiring variables to be explicitly quantified was thought to be helpful in detecting typographical errors. The use of ? to stand for an undefined value, type or structure was introduced in order to avoid the need for lookahead in parsing. The rationale behind the dynamic semantics given here is to maintain the ability to run programs as usual in the presence of undefined EML functors, structures, types and values; an exception will be raised only when an attempt is made to evaluate an undefined expression or access a component of an undefined structure.

At present, the mathematical basis of EML only supports the development of programs in a small purely functional subset of SML (excluding higher-order functions, polymorphism, references, exceptions, and input/output). As a consequence, the meaning of axioms which make use of features outside this subset is undefined. The syntax and static semantics of axioms reflect this restriction, but only up to a point:

**Higher-order functions:** Axioms may refer to higher-order functions but may not quantify over function types, although it is possible to get around this restriction as mentioned above (see rules 2 and 3). Function expressions are forbidden by excluding `fn` from the syntax of LogExp; however, there is no similar restriction on the *dec* part of a LogExp of the form `let` *dec* `in` *logexp* `end`.

**Polymorphism:** The use of polymorphism in axioms is unrestricted.

**References:** Direct use of the values `:=` and `ref` in axioms is forbidden (see rule 5). However, nothing prevents axioms from referring to other functions which cause side-effects.

**Exceptions:** Raising and handling exceptions in axioms is forbidden by excluding `raise` and `handle` from the syntax of LogExp. However, there is no similar restriction on the *dec* part of a LogExp of the form `let` *dec* `in` *logexp* `end`.

**Input/output:** The use of input/output functions in axioms is unrestricted.

Since the subset of SML programs which EML supports is expected to enlarge as time goes on (polymorphism appears to be unproblematic; higher-order functions and exceptions seem harder; handling references and input/output will be quite difficult) it was decided to take the path of least resistance and forbid some common things which seemed to be particularly easy to forbid while permitting the remainder for the present.

Implementors of EML who wish to provide users with the ability to experiment with axioms making use of assignment, exceptions etc. may choose to make LogExp a proper extension of Exp, and neglect to remove the values `:=` and `ref` from the environment in which axioms are elaborated (see rule 7). Users should however be aware that the meaning of such axioms is "officially" completely undefined.

## Acknowledgements

Thanks to Dave Matthews for helpful discussions. DS thanks Andrzej Tarlecki of the Polish Academy of Sciences for past and continuing collaboration on formal program development in Extended ML. The research reported here is supported by grant GR/E78463 of the Science and Engineering Research council. FdS is supported by a Brazilian government scholarship, CNPq process number 200459/88.0/CC.

## References

[HMT89] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML (version 3). Technical Report ECS-LFCS-89-81, LFCS, Computer Science Department, Edinburgh University, May 1989.

[Mac86] David MacQueen. Modules for Standard ML. Technical Report ECS-LFCS-86-2, LFCS, Computer Science Department, Edinburgh University, March 1986.

[San90] Donald Sannella. Formal program development in Extended ML for the working programmer. In *3rd BCS/FACS Workshop on Refinement*, Hursley Park, January 1990. To appear.

[ST85] Donald Sannella and Andrzej Tarlecki. Program specification and development in Standard ML. In *12th ACM Symp. on Principles of Programming Languages*, pages 67–77, New Orleans, January 1985.

[ST86] Donald Sannella and Andrzej Tarlecki. Extended ML: an institution-independent framework for formal program development. In *Workshop on Category Theory and Computer Programming*, pages 364–389, Guildford, September 1985. Springer Lecture Notes in Computer Science. Vol. 240, 1986.

[ST89] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: foundations and methodology. Technical Report ECS-LFCS-89-71, LFCS, Computer Science Department, Edinburgh University, February 1989. Extended abstract in *Intl. Colloq. on Current Issues in Programming Languages*, Barcelona, March 1989. Springer Lecture Notes in Computer Science. Vol. 352, 1989.

## A Appendix: Derived forms of Logical Expression

The derived forms for LogAtExp are constructed in the same way as those for AtExp. There are three derived forms of LogExp, presented in Figure 4. Each derived form is given with its equivalent form. As in [HMT89], we interpret each row of the table as a rewriting rule

$$\text{Derived form} \Longrightarrow \text{Equivalent form}$$

and these rules may be applied repeatedly to a phrase until it is transformed into a phrase of the bare language. The other derived forms for phrases of the Core Language remains as in the SML definition. Appendix B gives the full LogExp and LogAtExp grammar, including the derived forms.

**Logical Expressions** *logexp*

| Derived Form | Equivalent Form |
|---|---|
| $logexp_1$ `orelse` $logexp_2$ | `if` $logexp_1$ `then true else` $logexp_2$ |
| $logexp_1$ `andalso` $logexp_2$ | `if` $logexp_1$ `then` $logexp_2$ `else false` |
| $logexp_1$ `=>` $logexp_2$ | `if` $logexp_1$ `then` $logexp_2$ `else true` |

Figure 4: Derived forms of Logical Expressions

## B Appendix: Full Grammar for Logical Expression

The full grammar for LogExp consists of the grammar in Section 2.1 augmented by the derived forms of Appendix A. The full grammar for LogAtExp consists of the grammar in Section 2.1 augmented by derived forms as with AtExp in SML. The rules are shown in Figure 5.

| | | | |
|---|---|---|---|
| *logatexp* | ::= | *scon* | special constant |
| | | ⟨op⟩*longvar* | value variable |
| | | ⟨op⟩*longcon* | value constructor |
| | | { ⟨*logexprow*⟩ } | record |
| | | # *lab* | record selector |
| | | () | 0-tuple |
| | | ($logexp_1$ , ⋯ , $logexp_n$) | $n$-tuple, $n \geq 2$ |
| | | [$logexp_1$ , ⋯ , $logexp_n$] | list, $n \geq 0$ |
| | | ($logexp_1$ ; ⋯ ; $logexp_n$) | sequence, $n \geq 2$ |
| | | let *dec* in $logexp_1$ ; ⋯ ; $logexp_n$ end | local declaration, $n \geq 1$ |
| | | ( *logexp* ) | |
| *logexprow* | ::= | *lab* = *logexp* ⟨ , *logexprow*⟩ | logical expression row |
| *logexp* | ::= | *logatexp* | logical atomic |
| | | *logexp logatexp* | application (L) |
| | | $logexp_1$ *id* $logexp_2$ | infixed application |
| | | *logexp* : *ty* | typed (L) |
| | | exists *logmatch* | existential quantifier |
| | | forall *logmatch* | universal quantifier |
| | | $logexp_1$ andalso $logexp_2$ | conjunction |
| | | $logexp_1$ orelse $logexp_2$ | disjunction |
| | | $logexp_1$ => $logexp_2$ | implication (R) |
| | | if $logexp_1$ then $logexp_2$ else $logexp_3$ | conditional |
| *logmatch* | ::= | *logmrule* ⟨\| *logmatch*⟩ | |
| *logmrule* | ::= | *pat* => *logexp* | |

Figure 5: Full Grammar for EML Logical Expressions