# HaMLet

*To Be Or Not To Be Standard ML*

Version 2.0.1
2025/07/27

Andreas Rossberg
rossberg@mpi-sws.org

# Contents

# 1 Introduction

HaMLet is an implementation of Standard ML (SML'97), as defined in *The Definition of Standard ML* [MTHM97] – simply referred to as the *Definition* in the following text. HaMLet mainly is an interactive interpreter but also provides several alternative ways of operation, including a simple compiler to JavaScript [ES12]. Moreover, HaMLet can perform different phases of execution – like parsing, type checking, and evaluation – selectively. In particular, it is possible to execute programs in an untyped manner, thus exploring the space where "programs can go wrong".

## 1.1 Goals

The primary purpose of HaMLet is not to provide yet another SML system. Its goal is to be a faithful model implementation and a test bed for experimentation with the SML language semantics as specified in the Definition. It also might serve educational purposes. The main feature of HaMLet therefore is the design of its source code: it follows the formalisation of the Definition as closely as possible, only deviating where it is unavoidable. The idea has been to try to translate the Definition into an "executable specification". Much care has been taken to resemble names, operations, and rule structure used in the Definition and the *Commentary* [MT91]. Moreover, the source code contains references to the corresponding sections in the Definition wherever available.

On the other hand, HaMLet tries hard to get even the obscure details of the Definition right. There are some "features" of SML that are artefacts of its formal specification and are not straight-forward to implement. See the conclusion in Section 10 for an overview.

Efficiency was not a goal. Execution speed of HaMLet is not competitive in any way, since it naively implements the interpretative evaluation rules from the Definition. Usability was no priority either. The error messages given by HaMLet are usually rudimentary, to avoid complicating the implementation.

Since version 2, HaMLet stores the outcome of elaboration (type and environment information) into every node of a program's abstract syntax tree. We hope that this makes HaMLet more viable as a front-end for experimental language implementation work, for which the lack of access to type information in later phases turned out to be a major hurdle.

HaMLet has of course been written entirely in SML'97 and is able to bootstrap itself (see 2.7).

## 1.2 Bugs in the Definition

The Definition is a complex formal piece of work, and so it is unavoidable that it contains several mistakes, ambiguities, and omissions. Many of these are inherited from the previous language version SML'90 [MTH90] and have been documented accurately by Kahrs [K93, K96]. Those, which still seem to be present or are new to SML'97, are listed in Appendix A.

The general approach we take for resolving ambiguities and fixing bugs is doing it in the 'most natural' way. Mostly, this is obvious, sometimes it is not. Moreover, in cases where the Definition allows implementations some freedom (e.g. the choice of context taken into account to resolve overloading) we choose the most restrictive view, so that HaMLet only accepts those programs that ought to be portable across all possible implementations. The appendix discusses the resolutions we chose.

## 1.3 Related Work

HaMLet owes much of its existence to the first version of the ML Kit [BRTT93]. While the original Kit shared a similar motivation and a lot of inspiration came from that work, more recent versions moved the Kit into another direction. We hope that HaMLet can fill the resulting gap.

We also believe that HaMLet is considerably simpler and closer to the Definition. Moreover, unlike the ML Kit, it even implements the dynamic semantics of SML directly. On the other hand, HaMLet is probably less suited to serve as a library for real world projects, since no part of it has been tuned for efficiency in any way.

## 1.4 Copyright

Copyright of the HaMLet sources 1999-2025 by Andreas Rossberg.

The HaMLet source package includes portions of the SML/NJ library, which is copyright 1989-1998 by Lucent Technologies.

See `LICENSE.txt` files for detailed copyright notices, licenses and disclaimers.

HaMLet is free, and we would be happy if others experiment with it. Feel free to modify the sources in whatever way you want.

Please post any questions, bug reports, critiques, and other comments to

> `rossberg@mpi-sws.org`

# 2 Usage

## 2.1 Download

HaMLet is available from the following web page:

> `http://www.mpi-sws.org/hamlet/`

The distribution contains a tar ball of the SML sources and this documentation.

## 2.2 Systems Supported

HaMLet can be readily built with the following SML systems:

- SML of New Jersey (110 or higher) [NJ07]
- Poly/ML (5.0 or higher) [M07]
- Moscow ML (2.0 or higher) [RRS00]
- Alice ML (1.4 or higher) [AT06]
- MLton (20010706 or higher) [CFJW05]
- ML Kit (4.3.0 or higher) [K06]
- SML# (0.20 or higher) [ST07]

You can produce an executable HaMLet standalone with all systems. The first four also allow you to use HaMLet from within their interactive toplevel. This gives access to a slightly richer interface (see Section 2.6).

Other SML systems have not been tested, but should of course work fine provided they support the full language and a reasonable subset of the Standard Basis Library [GR04].

## 2.3 Prerequisites

HaMLet makes use of the Standard ML Basis Library [GR04][1]. In addition it uses two functors from the SML/NJ library [NJ98], namely `BinarySetFn` and `BinaryMapFn`, to implement finite sets and maps.

To generate lexer and parser, ML-Lex [AMT94] and ML-Yacc [TA00] have been used. The distribution contains all generated files, though, so you only have to install those tools if you plan to modify the grammar.

The SML/NJ library as well as ML-Lex and ML-Yacc are freely available as part of the SML of New Jersey distribution. However, the HaMLet distribution contains all necessary files from the SML/NJ library and the ML-Yacc runtime library. They can be found in the `smlnj-lib` subdirectory, respectively.[2]

## 2.4 Installation

To build a stand-alone HaMLet program, go to the HaMLet source directory and invoke one of the following commands:[3]

```
make with-smlnj
make with-mlton
make with-poly
make with-alice
make with-mosml
make with-mlkit
make with-smlsharp
```

depending on what SML system you want to compile with. This will produce an executable named `hamlet` in the same directory, which can be used as described in Section 2.5.[4]

The above `make` targets use the fastest method of building HaMLet from scratch. Most SML systems allow for incremental compilation that, after changes, only rebuilds those parts of the system that are affected. To perform an incremental built, use the following commands, respectively:[5]

```
make with-smlnj+
make with-alice+
make with-mosml+
make with-mlkit+
```

[1] Despite some incompatible changes between the two, HaMLet sources work with the latest specification of the Basis [GR04] as well as the previously available version [GR96].

[2] The sources of the SML/NJ library are copyrighted ©1989-1998 by Lucent Technologies. See `http://cm.bell-labs.com/cm/cs/what/smlnj/license.html` for copyright notice, license and disclaimer.

[3] Under DOS-based systems, Cygwin is required.

[4] If you are compiling with a version of Moscow ML prior to 2.10, then you need to patch the definition of `FIXES_mosml` as indicated in the `Makefile`.

[5] Currently, this only matters for Moscow ML and Alice ML, which employ batch compilers. The other systems either always build incrementally (SML/NJ, ML Kit), or do not support separate compilation at all (MLton, Poly/ML).

For other SML systems that are not directly supported, the makefile offers a way to build a single monolithic file containing all of the HaMLet modules:

```
make hamlet-bundle.sml
```

In principle, the resulting file should compile on all SML systems. In practice however, some might require additional tweaks to work around omissions or bugs in the provided implementation of the Standard Basis Library [GR04].[6]

After HaMLet has been built, you should be able to execute it as described in 2.5. Under Unixes, you have the option of installing HaMLet first:

```
make INSTALLDIR=mypath install
```

The default for `mypath` is `/usr/local/hamlet`. You should include your path in the `PATH` environment variable, of course.

## 2.5 Using the HaMLet Stand-Alone

After building HaMLet successfully with one of the SML systems, you should be able to start a HaMLet session by simply executing the command

`hamlet [-`*mode*`] [` *file* `...]`

The *mode* option you can provide, controls how HaMLet processes its input. It is one of

- `-p`: parsing mode (only parse input)
- `-l`: elaboration mode (parse and elaborate input)
- `-v`: evaluation mode (parse and evaluate input)
- `-x`: execution mode (parse, elaborate, and evaluate input)
- `-j`: JavaScript compilation mode (parse, elaborate, and compile input)

Execution mode is the default behaviour. Parsing mode will output the abstract syntax tree of the program in an intuitive S-expression format that should be suitable for further processing by external tools. Elaboration mode only type-checks the program, without running it.

Evaluation mode does not perform static analysis, so it can actually generate runtime type errors. They will be properly handled and result in corresponding error messages. Evaluation mode also has an unavoidable glitch with regard to overloaded constants: since no type information is available in evaluation mode, all constants will be assigned their default type. This can cause different results for some calculations. To see this, consider the following example:

```
0w1 div (0w2 * 0w128)                and
0w1 div (0w2 * 0w128) : Word8.word
```

Although both variants only differ in an added type annotation, the latter will have a completely different result – namely cause a division by zero and thus a `Div` exception (see also Appendix A.11). In evaluation mode, however, both are indistinguishable, and the second will actually behave like the first. You can still force calculation to be performed in 8 bit words by performing explicit conversions:

[6] Of the systems supported, SML/NJ, Moscow ML, the ML Kit, and SML# required such work-arounds, which appear as wrapper files for Standard Basis modules in the `fix` directory of the HaMLet source.

```
val word8 = Word8.fromLarge;
word8 0w1 div (word8 0w2 * word8 0w128);
```

Note that `LargeWord.word` = `word` in HaMLet.

JavaScript compilation mode will parse and elaborate, and then output JavaScript statements that are equivalent to the SML source. See Section 9 for more details.

If no file argument has been given you will enter an interactive session in the requested mode, just like in other SML systems. Input may spread multiple lines and is terminated by either an empty line, or a line whose last character is a semicolon. Aborting the session via Ctrl-D will exit HaMLet (end of file, Ctrl-Z on DOS-based systems).

Otherwise, all files are processed in order of appearance. HaMLet interprets the Definition very strictly and thus requires every source file to be terminated by a semicolon. A file name may be prefixed by `@` in which case it is taken to be an indirection file containing a white space separated list of other file names and expands to that list. Expansion is done recursively, i.e. the file may contain `@`-prefixed indirections on its own.

HaMLet currently provides a considerable part, but not yet the complete obligatory subset of the Standard Basis Library [GR04]. In particular, support for OS functionality is weak. Most basic types and corresponding operations are fully implemented, though. Support for the Basis Library can be turned off by passing the flag "`-b -`" on the command line (with an argument different from "–" this flag actually changes the load path for the library).

There are several things to note about HaMLet's output:

- Types and signatures are always fully expanded, in order to closely resemble the underlying semantic objects.
- Similarly, structure values are shown in full expansion.
- Signatures are annotated with the set of type names bound (as a comment).
- Similarly, the type name set of an inferred static basis is printed, though only elaboration mode.

### 2.6 Using HaMLet from within an SML System

You can also use HaMLet from within the interactive toplevel of a given SML system. This allows you to access the various modules described in the following sections of this document directly and experiment with them.

In most interactive SML systems – particularly HaMLet itself, see 2.7 – you should be able to load the HaMLet modules by evaluating

```
use "hamlet.sml";
```

As this requires recompiling everything, there are more comfortable ways for some particular systems:

- Under SML of New Jersey, it suffices to start SML/NJ in the HaMLet directory and evaluate[7]

  ```
  CM.make "sources.cm";
  ```

[7] In ancient versions of SML/NJ, i.e., before 110.20, the proper call would be `CM.make()`.

- Under Moscow ML, first go to the HaMLet directory and invoke

  ```
  make interactive-mosml
  ```

  Then start Moscow ML and type

  ```
  load "Sml";
  ```

Loading HaMLet into an SML session will create (besides others) a structure named `Sml`, providing the following signature:

```
signature SML =
sig
  val basisPath         : string option ref

  val parseString       : string -> unit
  val elabString        : string -> unit
  val evalString        : string -> unit
  val execString        : string -> unit
  val compileJSString   : string -> unit

  val parseFile         : string -> unit
  val elabFile          : string -> unit
  val evalFile          : string -> unit
  val execFile          : string -> unit
  val compileJSFile     : string -> unit

  val parseFiles        : string list -> unit
  val elabFiles         : string list -> unit
  val evalFiles         : string list -> unit
  val execFiles         : string list -> unit
  val compileJSFiles    : string list -> unit

  val parseSession      : unit -> unit
  val elabSession       : unit -> unit
  val evalSession       : unit -> unit
  val execSession       : unit -> unit
  val compileJSSession  : unit -> unit
end
```

The functions here come in four obvious groups:

- *x*`String` processes a program contained in the string given.
- *x*`File` processes a program contained in a file whose name is given.
- *x*`Files` processes a whole set of files in an incremental manner.
- *x*`Session` starts an interactive session, that can be exited by pressing Ctrl-D (end of file, Ctrl-Z on DOS-based systems).

Each call processes the program in the initial basis. For incremental processing, functions from the *x*`Files` or *x*`Session` group have to be used.

In each group there are five functions providing selective phases of execution:

- `parse`*X* just parses a program.
- `elab`*X* parses and elaborates a program.
- `eval`*X* parses and evaluates a program.
- `exec`*X* parses, elaborates, and evaluates a program.

- `compileJS`*X* parses, elaborates, and compiles a program to JavaScript.

These functions correspond to the different execution modes of the stand-alone HaMLet (see Section 2.5). They all print the resulting environments on `stdOut`, or a suitable error message on `stdErr` if processing does not succeed (parse functions just print `OK` on success). During processing of a file list or an interactive session, errors cause the current input to be skipped, but not abortion of the session.

Finally, provides `basisPath` gives a way to configure the directory from which HaMLet loads the Standard Basis Library. The default is `SOME "basis"`, as a path relative to the HaMLet binary. If set to `NONE`, no library is loaded, and only a bare minimum environment is provided, roughly resembling the initial basis from the Definition (plus the magic `use` function described in Section 8.5).

## 2.7 Bootstrapping

Since HaMLet has been written purely in strict SML'97, it is able to bootstrap itself. The file `hamlet.sml` provided in the source directory allows bootstrapping an interactive HaMLet session by starting the HaMLet stand-alone via

```
hamlet hamlet.sml wrap-hamlet.sml
```

Alternatively, the file can be `use`'d from within a HaMLet session. It will load all necessary modules enabling interactive use as described in 2.6.

Beware that loading the full Basis Library in the bootstrapped version will require a huge amount of virtual memory. If you are brave and have *lots* of memory and patience you can even try a second bootstrapping iteration from within a session on the bootstrapped HaMLet. Then, HaMLet not only type-checks itself but does also execute the type checker and evaluator itself. You should expect at least two orders of magnitude slowdown for each bootstrapping iteration, due to the naive interpretative evaluation.

## 2.8 Limitations

In its current version, HaMLet is not completely accurate with respect to some aspects of the SML language. The following list gives an overview:

- Parsing: The grammar in the Definition together with its informal disambiguation rules is rather over-ambitious. It is not possible to parse it with finite look-ahead, as required by conventional parsing technology – at least not without performing a major nightmare of grammar transformations first. Consequently, all existing SML implementations disallow some phrases that ought to be legal according to the Definition. The most obvious examples are mixtures of *fvalbind*s and `case` expressions like in

  ```
  fun f p1 = case e1 of p2 => e2
    | f p3 = e3
  ```

  No effort has been made to get this working in HaMLet. However, HaMLet is still more accurate than other SML implementations. For example, it parses the dreaded `where type` ... `and type` derived form for signature expressions correctly (see Section 4.4).

- Library: HaMLet does provide a significant portion of the Standard Basis Library, but it is not complete.

# 3 Overview of the Implementation

The implementation of HaMLet follows the Definition as closely as possible. The idea is to come as close as possible to the ideal of an executable version of the Definition. Where the sources deviate, they usually do so for one of the following reasons:

- the non-deterministic nature of some of the rules (e.g. guessing the right types in the static semantics of the core),
- the informal style of some parts (e.g. the restrictions in [4.11]),
- bugs or omissions in the Definition (see Appendix A).

We will explain non-trivial deviations from the Definition where appropriate.

The remainder of this document does not try to explain details of the Definition – the Commentary [MT91] is much better suited for that purpose, despite being based on the SML'90 Definition [MTH90]. Neither is this document a tutorial to type inference. The explanations given here merely describe the relation between the HaMLet source code and the formalism of the Definition. We makes reference to both, so it's best if you have Definition and HaMLet sources side by side. We use section numbers in brackets as above to refer to individual sections of the Definition. Unbracketed section numbers are cross references within this document.

Most explanations we give here will be rather terse and cover only general ideas without going into too much detail. The intention is that the source code speaks for itself for most part.

## 3.1 Structure of the Definition

The Definition specifies four main aspects of the SML language:

1. Syntax
2. Static semantics
3. Dynamic semantics
4. Program Execution

Syntax is the most conventional part of a language definition. The process of recognizing and checking program syntax is usually referred to as *parsing*. The static semantics is mainly concerned with the typing rules. The process of checking validity of a program with respect to the static semantics is called *elaboration* by the Definition. The dynamic semantics specifies how the actual *evaluation* of program phrases has to be performed. The last aspect essentially describes how the interactive toplevel of an SML system should work, i.e. how parsing, elaboration, and evaluation are connected. The complete processing of a program, performing all three aforementioned phases, is known as *execution*.

The four aspects are covered in separate chapters of the Definition. Further destructuring is done by distinguishing between core language and module language. This factorisation of the language specification is described in more detail in the preface and the first chapter of the Definition.

## 3.2 Modularisation

HaMLet resembles the structure of the Definition quite directly. For most chapters of the Definition there is a corresponding module, or a group of modules, implementing that aspect of the language, namely these are:

| | |
|---|---|
| Chapter 2 and 3 | `parse/Lexer`, `Parser`, `SyntacticRestrictions`*X* |
| Chapter 4 | `elab/ElabCore` |
| Chapter 5 | `elab/ElabModule` |
| Chapter 6 | `eval/EvalCore` |
| Chapter 7 | `eval/EvalModule` |
| Chapter 8 | `program/Program` |
| Appendix A | `parse/DerivedForms`*X* |
| Appendix B | `parse/Parser` |
| Appendix C | `elab/InitialStaticBasis` |
| Appendix D | `eval/InitialDynamicBasis` |
| Appendix E | `elab/OverloadingClass` (roughly) |

Most other modules implement objects and operations defined at the beginning of each of the different chapters, which are used by the main modules. The source of every module cross-references the specific subsections of the Definition relevant for the types, operations, or rule implementations contained in it.

Altogether, it should be quite simple to map particular HaMLet modules to concepts in the Definition and vice versa. To make the mapping as obvious as possible, we followed quite strict naming conventions (see 3.5). Each of the following sections of this document will cover implementation of one of the language aspects mentioned in 3.1. At the beginning of each section we will list all modules relevant to that part of the implementation.

As a rule, each source file contains exactly one signature, structure, or functor. The only exceptions are the files `Ids`*X*, `Syntax`, each containing a collection of simple functor applications, and the files containing the modules `Addr`, `ExName`, `Lab`, `Stamp`, `TyName`, `TyVar`, which also provide implementations of sets and maps of the corresponding objects.

## 3.3 Mapping Syntactic and Semantic Objects

The sets representing the different phrase classes of the SML syntax are defined inductively through the BNF grammars in the Definition. These sets are mapped to appropriate SML datatypes in obvious ways, using fields of type `option` for optional phrases.

All sets defining semantic objects in the Definition have been mapped to SML types as directly as possible:

| | |
|---|---|
| primitive objects (without structure) | abstract types |
| products ($A \times B$) | tuple types (`A * B`) |
| disjoint unions ($A \cup B$) | datatypes (`A of A \| B of B`) |
| $k$-ary products ($\cup_{k \geq 0} A^k$) | list types (`A list`) |
| finite sets ($\mathrm{Fin}(A)$) | instances of the `FinSet` functor |
| finite maps ($A \xrightarrow{\text{fin}} B$) | instances of the `FinMap` functor |

In some places, we had to relax these conventions somewhat and turn some additional types into datatypes to cope with mutual recursion between definitions. For example, environments are always rendered as datatypes.

Except for the primitive simple objects, no type definitions are abstract, i.e., type definitions representing structured sets from the semantics are always kept transparent. The sole reason is to allow the most literal translation of rules operating on semantic objects. Clearly, regarding this aspect, the HaMLet sources should not serve as an example for good modularisation practice...

### 3.4 Mapping Inference Rules

Usually, each group of inference rules in the Definition is implemented by one function. For rules of the form

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

the corresponding function has type

```
A * phrase -> A'
```

Each individual rule corresponds to one function clause. More specifically, an inference rule of the form:

$$\frac{A_1 \vdash \mathit{phrase}_1 \Rightarrow A'_1 \qquad \cdots \qquad A_n \vdash \mathit{phrase}_n \Rightarrow A'_n \qquad \text{side condition}}{A \vdash \mathit{phrase} \Rightarrow A'} \qquad (k)$$

maps to a function clause of the form:

```
elabPhraseClass args (A, phrase) =
(* [Rule k] *)
let
  val A1' = elabPhraseClass1(A1, phrase1)
  (* ... *)
  val An' = elabPhraseClassN(An, phraseN)
in
  if not(side condition) then
    error("message")
  else
    A'
end
```

Here, `args` denotes possible additional arguments that we sometimes need to pass around. There are exceptions to this scheme for rules that are not purely structural, e.g. rules 34 and 35 of the static semantics [4.10] are represented by one case only. Moreover, we deal slightly differently with the state and exception conventions in the dynamic semantics (see 6.3).

If one of a rule's premise is not met, an appropriate message is usually generated and an exception is raised through the `Error` module.

### 3.5 Naming Conventions

Structures and functors are named after the main type they define, the objects they generate, or the aspects of the Definition they implement (with one exception: the structure containing type `Int` is named `Inter` to avoid conflicts with the structure `Int` of the Standard Basis Library). The corresponding signatures are named accordingly.

Several structures come in groups, representing the separation of core and module language (and even the program layer). Orthogonal grouping happens for aspects similar in the static and dynamic semantics. The structure names reflect those connections in an obvious way, by including the words `-Core-`, `-Module-`, or `-Program-`, and `-Static-` or `-Dynamic-`.

Types representing sets defined in the Definition are always named after that set even if this conflicts with the usual SML conventions with respect to capitalisation. Functions are also named after the corresponding operation if it is defined in the Definition or the Commentary [MT91]. Variables are named as in the Definition, with Greek letters spelled out. Moreover, type definitions usually include a comment indicating how variables of that type will be named.

On all other occasions obvious names have been chosen, following conventions established by the Standard Basis Library [GR04] or the SML/NJ library [NJ98] where possible.

### 3.6 Side Effects

SML is not purely functional, and neither is the HaMLet implementation. It uses state whenever that is the most natural thing to do, or if it considerably simplifies code. State comes into play for the following:

- inside the lexer, to handle nested comments,
- inside the parser, to maintain the infix environment,
- in the abstract syntax tree, to annotate elaboration results,
- to generate time stamps, e.g. for type and exception names,
- in the representation of type terms, to allow destructive unification,
- during elaboration, to collect unresolved overloaded and flexible types,
- during evaluation, to maintain the program's state.

And of course, the code generated by Lex and Yacc uses state internally.

Other side effects are the output of error and warning messages in the Error structure.

## 4 Abstract Syntax and Parsing

### 4.1 Files

The following modules are related to parsing and representation of the abstract syntax tree:

| | |
|---|---|
| `syntax/` | |
| `Source` | representation of source regions |
| `Annotation` | representation of AST annotations |
| | |
| `IdFn` | generic identifier representation |
| `LongIdFn` | |
| `IdsCore` | instantiated identifier classes |
| `IdsModule` | |
| `TyVar` | type variable representation |
| `Lab` | label representation |
| `SCon` | special constants |
| | |
| `SyntaxCoreFn` | abstract syntax tree definition |
| `SyntaxModuleFn` | |
| `SyntaxProgramFn` | |
| `Syntax` | AST instantiations |
| | |
| `PPCore` | printing of core AST |

| | |
|---|---|
| `PPModule` | printing of module AST |
| `PPProgram` | printing of program AST |
| `PPSyntax` | auxiliary PP functions |
| | |
| `IdStatus` | identifier status |
| | |
| `parse/` | |
| `Lexer` | lexical analysis (via ML-Lex) |
| `LocLexer` | wrapper computing line locations |
| `Parser` | syntactical analysis (via ML-Yacc) |
| `Infix` | infix parser |
| `Parse` | parser plugging |
| | |
| `DerivedFormsCore` | derived forms transformation |
| `DerivedFormsModule` | |
| `DerivedFormsProgram` | |
| | |
| `BindingObjectsCore` | objects for binding analysis |
| `BindingObjectsModule` | |
| `BindingEnv` | operations on binding environment |
| `BindingContext` | operations on binding context |
| `BindingBasis` | operations on binding basis |
| | |
| `SyntacticRestrictionsCore` | verifying syntactic restrictions |
| `SyntacticRestrictionsModule` | |
| `SyntacticRestrictionsProgram` | |
| | |
| `elab/` | |
| `ScopeTyVars` | scoping analysis for type variables |

## 4.2 Abstract Syntax Tree and Annotations

The abstract syntax tree (AST) is split into three layers, corresponding to the SML core and module language and the thin program toplevel, respectively (modules `Grammar`$X$`Fn`). It represents the bare grammar, without derived forms. One notable exception has been made for structure sharing constraints, which are included since they cannot be handled as a purely syntactic derived form (see A.8). Infix directives [2.6] and application have been dropped from the core grammar, as they do not appear in the semantic rules of the Definition. However, we have to keep occurrences of the `op` keyword in order to do infix resolution (see 4.5).

Each identifier class is represented by its own abstract type. Most of them – except `TyVar` and `Lab`, which require special operations – are generated from the `IdFn` and `LongIdFn` functors.

Special constants are represented as strings containing the distinguished part of their lexical appearance – their actual values cannot be calculated before overloading resolution.

AST nodes consist of two parts: the actual syntax datatype and an annotation. The module `Annotation` defines an auxiliary datatype with an infix constructor to make construction and matching of nodes convenient:

```
datatype ('a, 'b) phrase = @@ of 'a * 'b annotation
```

With this type, the AST representation of an application expression can be written and pattern-matched as `APPExp(func, arg)@@A`, for example, where `A` is the annotation for this AST node.

The annotation itself is a static property list containing at least a `loc` property recording the character region in the original source text that this node corresponds to. The tail of each property list is functorised per phrase class in the AST definitions (functors `Syntax*Fn`).

The respective instantiations (file `Syntax`) define these property lists to contain a property `elab`, which will contain the outcome of elaboration to every node. For phrase types, it consists of the exact result returned by the respective elaboration rule, whereas identifier nodes are annotated with the classification these identifiers had in the environment at the respective point of use or definition (in particular, value identifiers are annotated with their respective polymorphic type scheme, whereas the expression node containing them is annotated with the instantiated type). Match and ValBind phrases have an additional property `exhaustive` that records whether the respective pattern is exhaustive.

More concretely, the source location of an AST node can be retrieved by invoking the function `Annotation.loc A` on its annotation value `A`; the static semantic object associated with a node via `Annotation.get(Annotation.elab A)` (after successful elaboration, otherwise the access will fail with an `Option` exception). The implementation of the JavaScript compiler (see Section 9) contains some examples of such usage.

## 4.3 Parsing and Lexing

Parser and lexer have been generated using ML-Yacc [TA00] and ML-Lex [AMT94] which are part of the SML/NJ distribution [NJ07]. The parser builds an abstract syntax tree using the syntax types described in Section 4.2.

Most parts of the parser and lexer specifications (files `Parser.grm` and `Lexer.lex`) are straightforward. However, we have to take some care to handle all those overlapping lexical classes correctly, which requires the introduction of some additional token classes (see comments in `Lexer.lex`). Nested comments are treated through a side-effecting counter for nesting depth.

A substantial number of grammar transformations is unavoidable to deal with LALR conflicts in the original SML grammar (see 4.4 and comments in `Parser.grm`). Some hacking is necessary to do infix resolution directly during parsing (see 4.5).

Semantic actions of the parser apply the appropriate constructors of the grammar types or a transformation function provided by the modules handling derived forms (see 4.6).

## 4.4 Grammar Ambiguities and Parsing Problems

ML-Yacc is a conventional LALR(1) parser generator. Unfortunately, the exact grammar given in the Definition, together with the disambiguation rules given in its [Appendix A] define a language that cannot be parsed by standard parsing technology, as it would require infinite look-ahead. The HaMLet parser is therefore incapable of handling all language constructs that are legal according to a strict reading of the Definition. The most annoying example of a problematic phrase is a `case` expression as right hand side of a function binding (see A.9). Most people consider this a bug on the side of the Definition. Consequently, we make no attempt to fix it in HaMLet. It could only be dealt with correctly either by horrendous grammar transformations or by some really nasty and expensive lexer hack [K93].

Disambiguation of expressions is left to ML-Yacc, we simply specify suitable keyword precedences. This seems to be the most appropriate thing to do, as the disambiguation rules in the Definition are ambiguous an contradictory by themselves (see A.9).

The SML grammar contains several other ambiguities on the declaration level (see A.1, A.2 and A.7). We resolve them in the 'most natural' ways:

- Semicolons are simply parsed as declarations or specifications, not as separators (cf. A.1).
- Sequential declarations and specifications are parsed left associative.
- Sharing specifications are also left associative, effectively at the same precedence level as sequential specifications.
- Core level declarations are reduced to structure declarations as soon as possible. This determines ambiguous `local` declarations (cf. A.2).

Several auxiliary phrase classes have been introduced to implement these disambiguations.

Some heavy transformations of the grammar are necessary to deal with the dreaded `where type`...`and type` derived form for signature expressions [Appendix A, Figure 19]: for every nonterminal $x$ that can end in a $sigexp$ and may be followed by another subphrase $y$ separated by the keyword '`and`' we had to introduce auxiliary nonterminals of the form

*x*__`AND`_*y*

whose semantic actions build two parts of the abstract syntax tree: the subtree for $x$ and the subtree for $y$.

Further grammar transformations are needed to cope with `as` patterns and datatype declaration vs. datatype replication.

## 4.5 Infix Resolution

Since ML-Yacc does not support attributes, and we did not want to introduce a separate infix resolution pass, the parser maintains an infix environment $J$ which is initialised and updated via side effects in the semantic actions of several pseudo productions. Applications – infix or not – are first parsed as lists of atomic symbols and then transformed by the module `Infix` which is invoked at the appropriate places in the semantic actions. The infix parser in that module is essentially a simple hand-coded LR Parser.

The parser is parameterised over its initial infix environment. After successful parsing it returns the modified infix environment along with the AST.

## 4.6 Derived Forms

To translate derived forms, three modules corresponding to the three grammar layers provide transformation functions that rewrite the grammatical forms to their equivalent forms, as specified in Appendix A of the Definition (modules `DerivedForms`$X$). These functions are named similar to the constructors in the AST types so that the parser itself does not have to distinguish between constructors of bare syntax forms and pseudo constructors for derived forms. To ensure that all node annotations are unique (given that they are stateful), some of the rewritings perfermed by these functions need to duplicate annotations accordingly.

The Definition describes the *fvalbind* derived form rather inaccurately. We made it a bit more precise by introducing several additional phrase classes (see A.9). Most of the parsing happens in the `Infix` module in this case, though.

Note that the structure sharing syntax is not a proper derived form since it requires context information about the involved structures (see A.8). It therefore was moved to the bare grammar.

### 4.7 Syntactic Restrictions

The BNF grammar given in the Definition actually specifies a superset of all legal programs, which is further restricted by a set of syntactic constraints [Section 2.9, 3.5]. The parser accepts this precise superset, and the syntactic restrictions are verified in a separate pass.

Unfortunately, not all of the restrictions given in the Definition are purely syntactic (see A.1). In general, it requires full binding analysis to infer identifier status and type variable scoping.

Checking of syntactic restrictions has hence been implemented as a separate inference pass over the whole program. The pass closely mirrors the static semantics. It computes respective binding environments that record the identifier status of value identifiers. For modules, it has to include structures, functors and signatures as well, because the effect of `open` relies on the environments they produce. Likewise, type environments are needed to reflect the effect of datatype replication. In essence, binding environments are isomorphic to interfaces in the dynamic semantics [Section 7.2]. As an extension, a binding basis includes signatures and functors. For the latter, we only need to maintain the result environment. Last, a binding context includes a set of bound type variables.

## 5 Elaboration

### 5.1 Files

The following modules represent objects of the static semantics and implement elaboration:

| | |
|---|---|
| `elab/` | |
| `StaticObjectsCore` | definition of semantic objects |
| `StaticObjectsModule` | |
| `TyName` | type names |
| | |
| `Type` | operations on types |
| `TypeFcn` | operations on type functions |
| `TypeScheme` | operations on type schemes |
| `OverloadingClass` | overloading classes |
| | |
| `StaticEnv` | environment instantiation |
| `Sig` | operations on signatures |
| `FunSig` | operations on functor signatures |
| `StaticBasis` | operations on basis |
| | |
| `ElabCore` | implementation of elaboration rules |
| `ElabModule` | |
| `Clos` | expansiveness check and closure |

| | |
|---|---|
| `CheckPattern` | pattern redundancy and exhaustiveness checking |

## 5.2 Types and Unification

Types are represented according to the mapping explained in Section 3.3 (see the modules `StaticObjectsCore` and `Type`). However, since type inference has to do unification (see 5.6), which we prefer to do destructively for simplicity, each type node actually is wrapped into a reference. A simple graph algorithm is required to retain sharing when cloning types. All other type operations besides unification have functional semantics.

In order to avoid confusion (cf. A.12) our type representation distinguishes undetermined types (introduced during type inference, see 5.6) from explicit type variables. This requires an additional kind of node in our type representation. Moreover, we have another kind of undetermined type node to deal with overloaded types (see 5.8). Finally, we need a third additional node that replaces undetermined types once they become determined, in order to retain sharing.

All operations on types have been implemented in a very straightforward way. To keep the sources simple and faithful to the Definition we chose not to use any optimisations like variable levels or similar techniques often used in real compilers.

## 5.3 Type Names

Type names (module `TyName`) are generated by a global stamp generator (module `Stamp`). As described in the Definition, they carry attributes for arity and equality.

To simplify the task of checking exhaustiveness of patterns type names have been equipped with an additional attribute denoting the *span* of the type, i.e. the number of constructors (see 5.12). For pretty printing purposes, we also remember the original type constructor of each type name.

## 5.4 Environment Representation

In order to share as much code as possible between the rather similar environments of the static and the dynamic semantics, as well as the interfaces Int in the dynamic semantics of modules, we introduce a functor `GenericEnvFn` that defines the representation and implements the common operations on environments.

Unfortunately, there exists a mutual recursion between environments and their range sets, in the static semantics (via TyStr) as well as in the dynamic semantics (via Val and FcnClosure). This precludes passing the environment range types as functor arguments. Instead, we make all environment types polymorphic over the corresponding range types. The instantiating modules (`StaticEnv`, `DynamicEnv`, and `Inter`) tie the knot appropriately.

## 5.5 Elaboration Rules

Elaboration implements the inference rules of sections [4.10] and [5.7] (modules `ElabCore` and `ElabModule`). It also checks the further restrictions in [4.11].

The inference rules have been mapped to SML functions as described in 3.4. They store the result of elaboration into the `elab` property of the respective AST annotations. To this end, the modules `Annotation` and `AnnotationElab` (the latter in file `Syntax`) specify two convenient operators, `->` and `|->`. This information can be retrieved from the AST as described in Section 4.2.

Additional arguments needed in some places of Core elaboration are encapsulated in an auxiliary record `deferred`: a flag indicating whether we are currently elaborating a toplevel declaration (in order to implement restriction 3 in [4.11] properly), a list of unresolved types (for overloading resolution and flexible records, see 5.8), and a list of `fn` *match*es (to defer checking of exhaustiveness until after overloading resolution, see 5.12 and 5.8). For modules, we pass down the equality attribute of type descriptions (see 5.10).

Note that most of the side conditions on type names could be ignored since they are mostly ensured by construction using stamps. We included them anyway, to be consistent and to have an additional sanity check. At some places these checks are not accurate, though, since the types examined can still contain type inference holes which may be filled with type names later. To be faithful, we hence employ time stamps on type names and type holes, such that violations of prior side conditions can be discovered during type inference, as we explain in the next section.

## 5.6 Type Inference

The inference rules for core elaboration are non-deterministic. For example, when entering a new identifier representing a pattern variable into the environment, rule 34 [4.10] essentially guesses its correct type. A deterministic implementation of type inference is the standard algorithm W by Damas/Milner [DM82]. Informally, when it has to guess a type non-deterministically it introduces a fresh type variable as a placeholder. We prefer to speak of undetermined types instead, since type variables already exist in a slightly different sense in the semantics of SML (cf. A.12).

Wherever an inference rule imposes an equality constraint on two types because the same meta-variable appears in different premises, the algorithm tries to unify the two types derived. After a value declaration has been checked, one can safely turn remaining undetermined types into type variables and universally quantify the inferred type over them, if they do not appear in the context. SML's value restriction does restrict this closure to non-expansive declarations, however [4.7, 4.8]. Note that (explicit) type variables can only be unified with themselves.

We use an imperative variant of the algorithm where unification happens destructively [C87], so that we do not have to deal with substitutions, and the form of the elaboration functions is kept more in line with the inference rules in the Definition.

Undetermined types are identified by stamps. They carry two additional attributes: an equality constraint, telling whether the type has to admit equality, and a time stamp, which records the relative order in which undetermined types and type names have been introduced. During unification with undetermined types we have to take care to properly enforce and propagate these attributes.

When instantiating type variables to undetermined types [4.10, rule 2], the undetermined type inherits the equality attribute from the variable. An undetermined equality type induces equality on any type it is unified with. In particular, if an undetermined equality type is unified with an undetermined non-equality type, equality is induced on the latter (function `Type.unify`).

Likewise, when a type is unified with an undetermined type, the latter's time stamp is propagated to all subterms of the former. That is, nested undetermined types inherit the time stamp if their own is not older already. Type names must always be older than the time stamp – unification fails, when a type name is encountered that is newer. This mechanism is used to prevent unification with types which contain type names that have been introduced *after* the undetermined type. For example, the snippet

```
let
  val r = ref NONE
  datatype t = C
in
  r := SOME C
end
```

must not type-check – the type of `r` may not mention `t` (otherwise the freshness side condition on names for datatypes [4.10, rule 17] would be violated). However, type inference can only find out about this violation at the point of the assignment expression. By comparing the time stamp of the undetermined type introduced when elaborating the declaration of `r`, and the stamp of the type name `t`, our unification algorithm will discover the violation.

More importantly, the mechanism is sufficient to preclude unification of undetermined types with *local* type names, as in the following example:

```
val r = ref NONE
functor F(type t; val x : t) =
struct
  val _ = r := SOME C
end
```

Obviously, allowing this example would be unsound.

To cope with type inference for records, we have to represent partially determined rows. The yet undetermined part of a row is represented by a special kind of type variable, a *row variable*. This variable has to carry the same attributes as an undetermined type, i.e. an equality flag and a time stamp, both of which have to be properly propagated on unification. See also Section 5.8.

## 5.7 Type Schemes

Type schemes represent polymorphic types, i.e. a type prefixed by a list of quantified type variables. The only non-trivial operation on type schemes is generalisation [4.5].

We implement the generalisation test via unification: in order to test for $\forall\alpha^{(k)}.\tau \succ \tau'$, we instantiate $\alpha^{(k)}$ with undetermined types $\tau^{(k)}$ and test whether $\tau[\tau^{(k)}/\alpha^{(k)}]$ can be unified with $\tau'$.

To test generalisation between type schemes, $\forall\alpha^{(k)}.\tau \succ \forall\alpha^{(k')}.\tau'$, we first skolemise the variables $\alpha^{(k')}$ on the right-hand side by substituting them with fresh type names $t^{(k')}$. Then we proceed by testing for $\forall\alpha^{(k)}.\tau \succ \tau'[t^{(k')}/\alpha^{(k')}]$ as described before.

Note that $\tau$ may contain undetermined types, stemming from "expansive" declarations for which the value restriction prevented generalisation. These have to be kept monomorphic, but naive unification might identify them with one of the skolem types $t^{(k')}$ (or a type containing one) – and hence effectively turn them into polymorphic types! For example, when checking the signature ascription in the following example,

```
signature S = sig val f : 'a -> 'a option end
structure X : S =
```

```
struct
    val r = ref NONE
    fun f x = !r before r := SOME x
end
```

the type inferred for the function `f` contains an undetermined type, the content type of `r`. It must be monomorphic, hence the type of `f` does not generalise the polymorphic type specified in the signature.[8] Comparison of the time stamps of the undetermined type and the newer type name generated during skolemisation of `'a` makes unification between the two properly fail with our algorithm.

## 5.8 Overloading and Flexible Records

Overloading is the least formal part of the Definition (see A.11). It is just described in an appendix, as special case treatment for a handful of given operators and constants. We tried to generalise the mechanism indicated in the Definition in order to have something a bit less ad hoc that smoothly integrates with type inference.

To represent type schemes of overloaded identifiers we allow type variables to be constrained with overloading classes in a type scheme, i.e. type variables can carry an overloading class as an additional optional attribute. When instantiated, such variables are substituted by overloaded type nodes, constrained by the same overloading class (constructor `Type.Overloaded`). When we unify an overloaded type with another, determined type we have to check whether that other type is a type name contained in the given overloading class. If yes, overloading has been resolved, if no there is a type error (function `Type.unify`).

When unifying two overloaded types, we have to calculate the intersection of the two overloading classes. So far, everything is pretty obvious. The shaky part is how to propagate the default types associated with the classes when we perform intersection.

We formalise an overloading class as a pair of its type name set and the type name being the designated default:

$$(T, t) \in \text{OverloadingClass} = \text{TyNameSet} \times \text{TyName}$$

Now when we have to intersect two overloading classes $(T_1, t_1)$ and $(T_2, t_2)$, there may be several cases. Let $T = T_1 \cap T_2$:

1. $T = \emptyset$. In this case, the constraints on the types are inconsistent and the program in question is ill-typed.
2. $T \neq \emptyset$ and $t_1 = t_2 \in T$. The overloading has (possibly) been narrowed down and the default types are consistent.
3. $T \neq \emptyset$ and $t_1 \neq t_2$ and $|\{t_1, t_2\} \cap T| = 1$. The overloading has been narrowed down. The default types differ but only one of them still applies.
4. $T \neq \emptyset$ and $|\{t_1, t_2\} \cap T| \neq 1$. The overloading could be narrowed down, but there is no unambiguous default type.

Case (3) is a bit subtle. It occurs when checking the following declaration:

```
fun f(x,y) = (x + y)/y
```

[8] Several SML implementations currently get this wrong, opening a soundness hole in their type checkers.

Both, `+` and `/` are overloaded and default to different types, but in this combination only `real` remains as a valid default so that the type of `f` should default to **real** $\times$ **real** $\rightarrow$ **real**.[9]

There are two ways to deal with case (4): either rule it out by enforcing suitable well-formedness requirements on the overloading classes in the initial basis, or handle it by generalising overloading classes to contain *sets* of default values (an error would be flagged if defaulting actually had to be applied for a non-singular set). We settled for the former alternative as it seems to be more in spirit with the Definition and it turns out that the overloading classes specified in the Definition satisfy the required well-formedness constraints.[10]

Consequently, we demand the following properties for all pairs of overloading classes $(T, t)$, $(T', t')$ appearing in a basis:

1. $t \in T$
2. $\mathrm{Eq}(T) = \emptyset \quad \vee \quad t$ admits equality
3. $T \cap T' = \emptyset \quad \vee \quad |\{t, t'\} \cap T \cap T'| = 1$

where $\mathrm{Eq}(T) = \{t \in T \mid t \text{ admits equality}\}$.

The reason for (1) is obvious. (2) guarantees that we do not loose the default by inducing equality. (3) ensures a unique default whenever we have to unify two overloaded types. (2) and (3) also allow the resulting set to become empty which represents a type error.

Defaulting is implemented by collecting a list of all unresolved types – this includes flexible records – during elaboration of value declarations (contained in the additional argument record `D : deferred`). Before closing an environment, we iterate over this list to default remaining overloaded types or discover unresolved flexible records. This implies that the context determining an overloaded type or flexible record type is the smallest enclosing core-level declaration of the corresponding overloaded identifier, special constant, or flexible record, respectively (cf. A.3 and A.11).

Special constants, which are also overloaded, have to be range-checked with respect to their resolved type [Section E.1]. For this purpose, the list of unresolved types can carry optional associated special constants. During defaulting we hence also do the corresponding range checking for all collected special constants.

## 5.9 Recursive Bindings and Datatype Declarations

Value bindings with `rec` and datatype declarations are recursive. The inference rules (26, 17 and 19) use the same environment $VE$ or $TE$ on the left hand side of the turnstile that is to be inferred on its right hand side.

To implement this we build a tentative environment in a first iteration that is not complete but already contains enough information to perform the actual inference in the second iteration. For recursive value bindings we insert undetermined types as placeholders for the actual types (and unify later), for datatype bindings we leave the constructor environments empty.

Datatype declarations bring an additional complication because of the side condition that requires $TE$ to maximise equality. This is being dealt with by first assuming equality for all new type names and later adjusting all invalid equality attributes in a fixpoint iteration, until all type structures respect equality (function `StaticEnv.maximiseEquality`).

[9] In fact, some SML implementations do not handle this case properly.

[10] A previous version of HaMLet used the latter alternative. It allows more liberal overloading but may lead to typing errors due to ambiguous overloading, despite the default mechanism. Moreover, in full generality it raises additional issues regarding monotonicity of overloading resolution when extending the library.

### 5.10 Module Elaboration

Like for the core language, the inference rules for modules are non-deterministic. In particular, several rules have to guess type names that have to be consistent with side conditions enforced further down the inference tree. However, most of these side conditions just ensure that type names are unique, i.e. fresh type names are chosen where new types are introduced. Since we create type names through a stamp mechanism, most of these side conditions are trivially met. The remaining cases are dealt with by performing suitable renaming of bound type names with fresh ones, as the Definition already suggests in the corresponding comments (module `ElabModule`).

The other remaining bits of non-determinism are guessing the right equality attribute for type descriptions, which is dealt with by simply passing the required attribute down as an additional assumption (function `ElabModule.elabTypDesc`), and for datatype specifications, which require the same fixpoint iteration as datatype declarations in the core (see 5.9).

### 5.11 Signature Matching

Signature matching is the most complex operation in the SML semantics. As the Definition describes, it is a combination of realisation and enrichment.

To match an environment $E'$ against a signature $\Sigma = (T, E)$ we first calculate an appropriate realisation $\varphi$ by traversing $E$: for all flexible type specifications in $E$ (i.e. those whose type functions are equal to type names bound in $T$) we look up the corresponding type in $E'$ and extend $\varphi$ accordingly. Then we apply the resulting realisation to $E$ which gives us the potential $E^-$. For this we just have to check whether it is enriched by $E'$ which can be done by another simple traversal of $E^-$ (functions `Sig.match` and `StaticEnv.enriches`).

The realisation calculated during matching is also used to propagate type information to the result environment of functor applications (rule 54, module `ElabModule`). A functor signature has form $(T_1)(E_1, (T_1')E_1')$. To obtain a suitable functor instantiation $(E'', (T')E')$ for rule 54 we simply match the environment $E$ of the argument structure to the signature $(T_1)E_1$ which gives $E''$ and a realisation $\varphi$. We can apply $\varphi$ to the functor's result signature $(T_1')E_1'$ to get – after renaming all $t \in T_1'$ to fresh names $t' \in T'$ – the actual $(T')E'$ appearing in the rule.

### 5.12 Checking Patterns

Section [4.11], items 2 and 3 require checking exhaustiveness and irredundancy of patterns (module `CheckPattern`). The basic idea of the algorithm is to perform *static matching*, i.e. to traverse the decision tree corresponding to a match and propagate information about the value to be matched from the context of the current subtree. The knowledge available on a particular subterm is described by the `description` type. Moreover, a `context` specifies the path from the root to the current subtree.

The algorithm is loosely based on [S96], where more details can be found. To enable this algorithm, type names carry an additional attribute denoting their *span*, i.e. the number of constructors the type possesses (see 5.3). We extend the ideas in the paper to cover records (behave as non-positional tuples), exception constructors (have infinite span), and constants (treated like constructors with appropriate, possibly infinite span). Note that we have to defer checking of patterns until overloading resolution for contained constants has been performed – otherwise we will not know their span.

A context description is not simply a list of constructor applications to term descriptions as in the paper, but separates constructor application from record aggregation and uses a nested definition. Instead of lists of negative constructors (and constants) we use sets for descriptions. Record descriptions are maps from labels to descriptions.

During traversal we construct two sets that remembers the region of every match we encountered, and every match we reached. In the end we can discover redundant matches by taking the difference of the sets. Non-exhaustiveness is detected by remembering whether we reached a failure leaf in the decision tree.

In the case of exception constructors, equality can only be checked on a syntactic level. Since there may be aliasing this is merely an approximation (see A.3).

There is a problem with the semantics of sharing and `where` constraints, which allow inconsistent datatypes to be equated (see A.3). In this case, no meaningful analysis is possible, resulting warnings may not make sense. There is nothing we can do but ignore this problem.

# 6 Evaluation

## 6.1 Files

Objects of the dynamic semantics and evaluation rules are implemented by the following modules:

| | |
|---|---|
| `eval/` | |
| `DynamicObjectsCore` | definition of semantic objects |
| `DynamicObjectsModule` | |
| `Addr` | addresses |
| `ExName` | exception names |
| `BasVal` | basic values |
| `SVal` | special values |
| | |
| `Val` | operations on values |
| `State` | operations on state |
| | |
| `DynamicEnv` | operations on environments |
| `Inter` | operations on interfaces |
| `DynamicBasis` | operations on basis |
| `IntBasis` | operations on interface basis |
| | |
| `EvalCore` | implementation of evaluation rules |
| `EvalModule` | |

## 6.2 Value Representation

Values are represented by a datatype corresponding to what is defined in Section [6.3] of the Definition (module `DynamicObjectsCore`). Special values are simply represented by the corresponding SML types (module `SVal`). Currently, only the default types and

`Word8.word` are implemented, which represents the minimum requirement of the Standard Basis.

Basic values are simply represented by strings (module `DynamicObjectsCore`). However, the only basic value defined in the Definition is the polymorphic equality =, everything else is left to the library. Consequently, the implementation of the APPLY function in module `BasVal` only handles =. For all other basic values it dispatches to the `DynamicLibrary` module, which provides an extended, library-specific version of the APPLY function (see Section 8).

The special value FAIL, which denotes pattern match failure, is not represented directly but has rather been defined as an exception (see 6.3).

## 6.3 Evaluation Rules

The rules of the dynamic semantics have been translated to SML following similar conventions as for the static semantics (see Section 3.4). However, to avoid painfully expanding out all occurrences of the state and exception conventions, we deal with state and exceptions in an imperative way. State is not passed around as a functional value but rather as a reference to the actual state map (module `State`) that gets updated on assignments. This avoids threading the state back with the result values. Exception packages (module `Pack`) are not passed back either, but are rather transferred by raising a `Pack` exception. Similarly, FAIL has been implemented as an exception.

So state is implemented by state and exceptions by exceptions – not really surprising. Consequently, rules of the form

$$s, A \vdash \mathit{phrase} \Rightarrow A'/p, s'$$

become functions of type

```
State ref * A * phrase -> A′
```

which may raise a `Pack` exception – likewise for rules including FAIL results. We omit passing in the state where it is not needed. This way the code follows the form of rules using the state and exception conventions as close as possible (modules `EvalCore` and `EvalModule`).

Failure with respect to a rule's premise corresponds to a runtime type error. This may actually occur in evaluation mode and is flagged accordingly.

Evaluation of special constant behaves differently in execution and evaluation mode. In the former, constants will have been annotated with a proper type name by overloading resolution (see 5.8). In evaluation mode this annotation is missing and the function `valSCon` will assume the default type of the corresponding overloading class, respectively. This implies that the semantics may change (see 2.5).

Note that the rules 182 and 184–186 of the dynamic semantics for modules contain several errors (see A.6).

# 7 Toplevel

## 7.1 Files

The remaining modules implement program execution and interactive toplevel:

| | |
|---|---|
| `exec/` | |
| `Basis` | the combined basis |
| `Program` | implementation of rules for programs |
| | |
| `elab/` | |
| `ElabProgram` | separate elaboration |
| `eval/` | |
| `EvalProgram` | separate evaluation |
| | |
| `parse/` | |
| `InitialInfixEnv` | initial environments |
| `elab/` | |
| `InitialStaticEnv` | |
| `InitialStaticBasis` | |
| `eval/` | |
| `InitialDynamicEnv` | |
| `InitialDynamicBasis` | |
| | |
| `infrastruture/` | |
| `PrettyPrint` | pretty printing engine |
| `PPMisc` | auxiliary pretty printing functions |
| `elab/` | |
| `PPType` | pretty printing of types |
| `PPStaticEnv` | ... static environment |
| `PPStaticBasis` | ... static basis |
| `eval/` | |
| `PPVal` | ... values |
| `PPDynamicEnv` | ... dynamic environment |
| `PPDynamicBasis` | ... dynamic basis |
| `exec/` | |
| `PPEnv` | ... combined environment |
| `PPBasis` | ... combined basis |
| | |
| `main/` | |
| `Sml` | main HaMLet interface |
| `Main` | wrapper for stand-alone version |

## 7.2 Program Execution

The module `Program` implements the rules in Chapter 8 of the Definition. It follows the same conventions as used for the evaluation rules (see 3.4 and 6.3).

In addition to the 'proper' implementation of the rules as given in the Definition (function `execProgram`) the module also features two straightforward variations that suppress evaluation and elaboration, respectively (`elabProgram` and `evalProgram`).

Note that a failing elaboration as appearing in rule 187 corresponds to an `Error` exception. However, in evaluation mode, any `Error` exception will instead originate from a runtime type error.

The remaining task after execution is pretty printing the results. We use an extended version of a generic pretty printer proposed by Wadler [W98] which features more sophisticated grouping via *boxes* (modules `PrettyPrint` and `PP`$xxx$).

In addition to the rule implementations in module `Program`, which implement interactive execution as prescribed by the Definition, we also provide two separate modules `ElabProgram` and `EvalProgram`, which implement separate elaboration and evaluation for program phrases in a manner consistent with the rules for the rest of the language. These modules are not used for program execution by HaMLet itself, but intended as a building blocks for language implementations on top of HaMLet. In particular, HaMLet's JavaScript compilation mode (Section 9) uses `ElabProgram`.

### 7.3 Plugging

The `Sml` module sets up the standard library (see Section 8), does all necessary I/O interaction and invokes the parser and the appropriate function in module `Program`, passing the necessary environments.

After processing the input itself the functions in the `Sml` module process all files that have been entered into the `use` queue during evaluation (see 8.5). That may add additional entries to the queue.

The `Main` module is only needed for the stand-alone version of HaMLet. It parses the command line and either starts an appropriate session or reads in the given files.

## 8 Library

### 8.1 Files

The library only consists of a hook module and the library implementation files written in the target language:

| | |
|---|---|
| `lib/` | |
| `StaticLibrary` | primitive part of the library (static definitions) |
| `DynamicLibrary` | primitive part of the library (dynamic definitions) |
| `Use` | `use` queue |
| `basis/` | the actual library modules |

### 8.2 Language/Library Interaction

The Definition contains several hooks where it explicitly delegates mechanics to the library:

- the set BasVal of basic values and the APPLY function [6.4],
- the initial static basis $B_0$ and infix status [Appendix C],
- the initial dynamic basis $B_0$ [Appendix D],

- the basic overloading classes Int, Real, Word, String, Char [E.1].

Realistically, it also would have to allow extending the sets SVal [6.2] and Val [6.3], and enable the APPLY function to modify the program state (cf. A.5). HaMLet currently only extends SVal, while other library types are mapped to what is there already (see 8.4). All respective library extensions are encapsulated into a pair of modules `StaticLibrary` and `DynamicLibrary` that define the parts of these objects that are left open by the Definition.

However, we split up implementation of the overall library into two layers:

- the *primitive* layer contains all that cannot be self-hosted in the implemented SML,
- the *surface* layer defines the actual library.

Let us call the instance of SML that HaMLet implements the *hosted* language, while the SML universe *in* which HaMLet is implemented the *hosting* language. Many library entities are definable within the hosted language itself, e.g. the standard `!` function. There are basically three reasons that can force us to make an entity primitive:

- its behaviour cannot be implemented out of nowhere (e.g. I/O operations),
- it is dependent on system properties (e.g. numeric limits), or
- it possesses a special type (e.g. overloaded identifiers).

The `StaticLibrary` and `DynamicLibrary` modules define everything in the hosting language that has to be primitive (see 8.3), while the rest is implemented within the hosted language in the modules inside the `basis` directory (see 8.6). These modules have to make assumptions about what is defined by the primitive library modules, so that both layers should be seen in conjunction.

## 8.3 Primitives

Primitive operations are implemented by means of the APPLY function. Most of them just fall back to the corresponding operations of the host system.[11] We only have to unpack and repack the value representation and remap possible exceptions. Overloaded primitives have to perform a trivial type dispatch.

Despite implementing a large number of primitives, the static and dynamic basis exported does only contain a few things:

- the `vector` type,
- all overloaded functions,
- the exceptions used by primitives,
- the function `use`.

(Non-toplevel primitive types and exceptions, like `Word8.word` and `IO.Io`, are wrapped into their residuent structures.) Everything else can be defined from these in the hosted language.

To enable the hosted language to bind the basic values defined by the primitive library, we piggy-back the `use` function. Its dynamic semantics is overloaded and in the static basis exported by the `StaticLibrary` module it is given type $\alpha \to \beta$. Applying it to a record of type $\{\mathtt{b} : \mathtt{string}\}$ will return the basic value denoted by the string `b` Primitive constants of type $\tau$ are available as functions $\mathtt{unit} \to \tau$. Once all primitives are extracted, the self-hosted library implementation restricts `use` to its proper safe type through a type-annotated rebinding.

[11]Unfortunately, most SML implementations lack a lot of the obligatory functionality of the Standard Basis Library. To stay portable among systems we currently restrict ourselves to the common subset.

### 8.4 Primitive Library Types

The dynamic semantics of the Definition do not really allow the addition of arbitrary library types – in general this would require extending the set Val [6.3]. Moreover, the APPLY function might require access to the state (see A.5).

But we can at least encode vectors by abusing the record representation. Arrays can then be implemented on top of vectors and references within the target language. However, this has to make their implementation type transparent in order to get the special equality for arrays.

I/O stream types can only be implemented magically as indices into a stateful table that is not captured by the program state defined in [6.3].

### 8.5 The `use` Function

The 'real' behaviour of `use` is implemented by putting all argument strings for which it has been called into a queue managed by module `Use`. The `Sml` module looks at this queue after processing its main input (see 7.3).

The argument strings are interpreted as file paths, relative paths being resolved with respect to the current working directory before putting them into the queue. The function reading source code from a file (`Sml.fromFile`) always sets the working directory to the base path of the corresponding file before processing it. This way, `use` automatically interprets its argument relative to the location of the current file.

### 8.6 Library Implementation

The surface library is loaded on startup. The function `Sml.loadLib` just silently executes the file `basis/all.sml`. This file is the hook for reading the rest of the library, it contains a bunch of calls to `use` that execute all library modules in a suitable order. Note that the library files always have to be *executed*, even if HaMLet is just running in parsing or elaboration mode – otherwise the contained `use` applications would not take effect.

The library modules themselves mostly contain straightforward implementations of the structures specified in the Standard Basis Manual [GR04]. Like the implementation of the language, the library implementation is mostly an executable specification with no care for efficiency. All operations not directly implementable and thus represented as primitive basic values are bound via the secret functionality of the `use` function (see 8.3).

## 9 Compilation to JavaScript

Since version 2, HaMLet provides an additional mode of operation under which the input is translated to equivalent JavaScript code (compatible with EcmaScript edition 3 and later [ES12]). The main motivation for adding this mode was to provide an example of a simple compiler that uses HaMLet's parser and elaborator as its front-end, and especially, uses the elaboration annotations on the AST (cf. Section 4.2). The JavaScript compiler utilises this elaboration information in several ways:

- to statically resolve uses of overloaded operators,
- to do arity conversion for functions and constructors,
- to distinguish variable bindings from constructors in patterns,

- to distinguish datatype from exception constructors in patterns,
- to compile `open` declarations and datatype replications,
- to compile scoping of `local` and `abstype` declarations,
- to detect shadowing between sequential declarations.

Some of these uses are described below.

## 9.1 Usage

JavaScript compilation mode is activated from the command line via the flag `-j` (cf. Section 2.5). With embedded usage it can be invoked through one of the respective `compileJS`*xxx* functions from the `Sml` module (cf. Section 2.6).

HaMLet then outputs JavaScript source code to the standard output, which can be redirected to a file and be executed in a browser console or a JavaScript shell. The generated code assumes presence of the runtime functionality provided with the file `runtime.js`, and a (translation of) the Standard Basis Library. Invoking

```
make js
```

creates a file `basis.js` in the HaMLet root directory that bundles both of these (the displayed shadowing warnings can be safely ignored in this case). This file either has to be loaded into the browser or JavaScript shell first, or it can be prepended to the output generated for the user program.

HaMLet can bootstrap itself on top of JavaScript: invoking

```
make hamlet.js
```

creates a bundled implementation of HaMLet in JavaScript (including the contents of `basis.js`). However, it may be necessary to bump the stack size limit of the respective JavaScript environment to actually execute the generated file.

Some caveats apply when translating SML to JavaScript:

- The compiler needs to use some coding tricks to implement shadowing within the same declaration scope. In interactive mode, this is not always possible, and a warning is generated if the compiler discovers toplevel shadowing (see 9.3 for details).

- The standard JavaScript execution environment in the browser does not provide any real I/O capabilities. The runtime library emulates `TextIO`'s standard *output* (`stdOut`) through the `console.log` function. It also emulates a simple file system in memory to fake file I/O and provide the functionality from `OS.FileSys`. However, standard *input* from a user can only be simulated statically, by defining a fixed string for the contents of `TextIO.stdIn` (see 9.4 for details).

- JavaScript does not (yet) support tail-call optimisation. Our simple translation makes no attempt to simulate it.

The former two issues are described in more detail in Section 9.3.

In general, our translation is more a proof of concept than an efficient compiler. For example, it will naively create a lot of nested closures to directly simulate SML scoping, instead of renaming identifiers in a whole-program manner. For a more production-quality compiler, see e.g. SMLtoJs [E08], which is based on the ML Kit [BRTT93].

## 9.2 Files

The compiler isn't particularly voluminous:

| | |
|---|---|
| `compile-js/` | |
| `JSSyntax` | JavaScript kernel AST |
| `PPJS` | JavaScript pretty printer |
| | |
| `IdSetCore` | computation of free and bound identifiers |
| `IdSetModule` | |
| | |
| `JSTranslateSCon` | compilation |
| `JSTranslateId` | |
| `JSTranslateCore` | |
| `JSTranslateModule` | |
| `JSTranslateProgram` | |
| `CompileJS` | main compiler entry point |
| | |
| `runtime.js` | runtime library |

## 9.3 Translation

To a large extent, the compilation scheme we implement is straightforward. Most types of SML values have fairly direct counterparts in JavaScript:

- Values of *primitive types* are translated to the respective JavaScript types: integers, words and reals to numbers; strings and characters to strings. The respective operations over these types are implemented accordingly.
- The `unit` value `()` is mapped to `undefined` in JavaScript.
- Other *records* are translated directly to JavaScript objects with the respective property names. In particular, that results in *tuples* being represented as JavaScript arrays, e.g., the SML tuple `(3, 4)` corresponds to the JS array `[3, 4]`. Projection hence simply becomes property access, with numeric labels shifted by 1.
- *Functions* obviously map to JavaScript functions. However, functions whose argument is an $n$-ary tuple will be converted to an $n$-ary function. See below for more details.
- *Datatype values* are represented as either strings (for nullary constructors) or objects (for constructors with arguments). In the former case, the string simply contains the constructor's name, in the latter the object has one property named after the constructor, which carries the constructor argument. For example, `NONE` becomes `'NONE'`, and `SOME 3` becomes `{'SOME': 3}`,and `1::2::nil`, which is short for `::(1, ::(2, nil))` in SML, becomes `{'::': [1, {'::': [2, 'nil']}]}`. Pattern matching simply does the respective string comparison, or checks for the presence of a property with the constructor's name (via JavaScript's `in` operator).
  *Constructors* themselves (i.e., $n$-ary constructors) are represented by functions, in order to enable using them in a first-class fashion. Like for other functions, tupled arguments will be flattened into an $n$-ary argument list.
- *Exceptions* are represented as either instances of JavaScript's `Error` function (for nullary exceptions) or as instances of a new function, named after the exception, whose prototype is `Error` (for constructors with arguments). In the former case,

the constructor's name is used as the error name, in the latter, the error value will have a property `"of"` carrying the argument (in contrast to datatype constructors we cannot use the constructor name, since that can be aliased). Tupled arguments to exception functions will again be flattened into an $n$-ary argument list. Pattern matching a nullary exception is a simple identity comparison, whereas matching other exceptions becomes an `instanceof` test.

- *References* are represented as objects of the form `{ref: x}`.
- *Vectors* and *arrays* are mapped to JavaScript arrays.
- *Structures* and *functors* become objects and functions in the obvious manner. Signature annotations yield a new object with only the exported fields.
- Finally, *types* and *signatures* are completely erased by the translation.

SML identifiers are mostly mapped to JavaScript identifiers directly. However, in some cases some extra work is required:

- JavaScript keywords are escaped with a leading underscore.
- The tick `'` in alphanumeric identifiers is replaced by `$`.
- Symbolic identifiers are translated into clear text, escaped and separated by underscores. For example, `:=` becomes `_colon_equal`.
- To separate structure and functor name spaces from values, identifiers in these spaces are escaped with `$` and `$$`, respectively. (Types and signatures are erased, so don't matter.)

Note that identifiers starting with an underscore _ or tick `'` are not valid SML, so we can make liberal use of both initial underscores or `$` for escaping.

Most other language constructs can also be translated fairly directly. However, there are two aspects to the translation that require a bit more work: arity conversion for functions and non-trivial scoping. We sketch these briefly in the following.

### Arity Conversion

To produce the most natural JavaScript functions in the common case, we translate all functions that have a tuple type for argument (including `unit` as the 0-tuple) to a JavaScript function with the respective number of arguments. Analogously, all calls with a tuple as argument are converted to a call with an argument list.

Unfortunately, these transformations induce extra complications due to polymorphism and abstract types:

- When calling a function with an abstract type for argument, this type may actually be implemented as a tuple, and consequently, the function may expect an unboxed argument list. In these cases, we introduce a runtime check for a tuple (i.e., an array on the Javacript side) before the call, and if so, call the function via JavaScript's `apply` method, which performs the unboxing (after slicing off the hole at position 0 of the array representing the tuple).
  For example, given the functions `f : t -> unit`, where `t` is an abstract type, the call `f x` generates the following JavaScript:

  ```
  (_SML._isTuple(x) ? f.apply(undefined, x) : f(x))
  ```

  where `_SML._isTuple` is a function from the runtime library (see Section 9.4) implemented as follows:

```
function _isTuple(x) {
  return x instanceof _JS.Array
    && !(x instanceof _SML.Vector._constructor);
}
```

- Dually, if a function is defined with a single parameter of abstract type (including a polymorphic type variable), it might actually be instantiated to a tuple. Hence, for such polymorphic functions we insert a transformation that boxes the `arguments` array into a tuple if its length is not 1 at runtime (where 'boxing' the empty array produces the `undefined` value).
  For example,

  ```
  fun id x = x
  ```

  will be translated into

  ```
  var id = function() {
    var x = _SML._tuplifyArgs(arguments);
    return x;
  }
  ```

  to make sure that a call like `id(3, 4)` actually returns a tuple. Here, the function `_SML._tuplifyArgs` is again part of our runtime, and defined as:

  ```
  function _tuplifyArgs(args) {
    return args.length <= 1 ? args[0] : [].slice.call(args);
  }
  ```

  If the arguments object consists of multiple arguments, this function converts them into our representation of a tuple (see above). Otherwise, if there is only one value, it is returned. Or, if the arguments array happens to be empty, `x[0]` produces the value `undefined`, as desired.

Performing these dynamic conversions avoid the need for monomorphisation and defunctorisation that would otherwise be necessary to fully handle arity conversion.

**Scoping**

A second problem is the lack of suitable scoping constructs in JavaScript, which complicates the translation of SML declarations. For starters, JavaScript has no `let`-like construct, so we have to simulate local scopes through a function abstraction. For example,

```
3 + let val x = 1; val y = 2 in x + y end
```

becomes

```
3 + (function(){ var x = 1; var y = 2; return x + y })()
```

But SML also allows *shadowing* within a single scope. We want to avoid renaming, so we deal with these cases by splitting a scope where an overlap is introduced, such that the shadowing happens in a nested (function) scope on the JavaScript side. From there, all non-shadowed variables are returned packed up as an object, which is then "opened" in the original scope. For example,

```
val a = 1
val f = fn() => a
val a = 2
val b = (f(), a)  (* b = (1, 2) *)
```

is translated to the following JavaScript:

```
var _x1 =
  (function() {
    var a = 1;
    return (function() {
        var f = function() { return a; };
        var a = 2;
        var b = [f(), a];  // b == [1, 2]
        return {a: a, b: b, f: f};
      })();
  })();
var a = _x1.a;
var b = _x1.b;
var f = _x1.f;
```

A similar scheme is used to translate `local` and `abstype` declarations.

One caveat with this transformation is that it requires the whole extent of the remaining scope to be known, so that it can be wrapped into the auxiliary functions. Consequently, it does not generally work in the toplevel scope, which is allowed to be extended and compiled incrementally. Every *program* phrase will create global bindings, which in JavaScript are mutable. If a *program* shadows a previous binding, its translation hence will *overwrite* (i.e., mutate) that binding. For example, the slight variation of the previous examples with extra semicolons (which turns each declaration into a separate *program* phrase),

```
val a = 1;
val f = fn() => a;
val a = 2;
val b = (f(), a);  (* b = (1, 2) *)
```

simply becomes

```
var a = 1;
var f = function() { return a; };
var a = 2;
var b = [f(), a];  // b == [2, 2] !
```

which obviously is incorrect.

A warning is generated if the compiler discovers toplevel shadowing that could potentially yield to incorrect code. To avoid this issue, either avoid top-level shadowing, or do not use semicolons as separators. (If we did not compile the toplevel that way, then it would not be possible to concatenate translated programs.)

## 9.4 Runtime

In the interpreted mode of execution, HaMLet self-hosts all SML language primitives in SML itself, as we described in Section 8. These primitives are made available to the hosted language via the `use` function. When compiling to JavaScript, the same primitives (including the `use` function itself) must be implemented in JavaScript.

The implementation of this runtime library lives in `runtime.js`. It contains the initial dynamic basis, i.e., all primitive toplevel values, and an object `_SML` that hosts all internal runtime functionality. In particular, all library primitives accessible via the `use` function are located in nested objects (corresponding to library structures) of `_SML`. In addition, it contains a couple of internal helpers and internal store, distinguished by attribute names starting with an underscore.

Most of the runtime implementation is straightforward, only I/O requires jumping through some hoops:

- The standard JavaScript execution environment is the browser, and hence does not provide any real I/O capabilities. The runtime library emulates `TextIO.stdOut` and `TextIO.stdErr` through the `console.log` function (with extra buffering).

  `TextIO.stdIn` cannot be emulated directly, but a fixed input can be simulated by the runtime with its internal `_SML.TextIO._stdIn.content` reference. By setting it to an appropriate string before running the generated program, the runtime will pretend actual input. For example, assigning

  ```
  _SML.TextIO._stdIn.content =
    "input line 1\ninput line 2\ninput line\n";
  ```

  (in JavaScript land) will make the runtime behave as if the user had input three respective lines and then generated EOF (e.g., as if pressing Ctrl-D on a Unix system).

- A similar work-around is available for emulating the command line. From within JavaScript, one can set the values of `_SML.CommandLine._name` (initially, the string `"hamlet"`) and `_SML.CommandLine._arguments` (by default an empty array) to customise the results of the respective functions from the `CommandLine` structure.

- The runtime also simulates a simple file system in memory. This is initially empty, consisting only of the root directory. A directory structure can easily be created in SML. It can also be pre-configured on the JavaScript side, by appropriate calls to `_SML.OS.FileSys.mkdir`, and can then be pre-populated with files via the convenience function `_SML.OS.FileSys.file`. For example, the JavaScript calls

  ```
  _SML.OS.FileSys.mkDir("a");
  _SML.OS.FileSys.mkDir("a/b");
  _SML.file("foo.txt", "Hello");
  _SML.file("a/bar.txt", "world\n");
  _SML.file("a/b/baz.bin", "\x53\x4d\x4c");
  ```

  would create a simple directory structure with three files.

Given these hacks, it should be sufficiently easy to run SML programs with simple simulated I/O in the browser.

# 10 Conclusion

HaMLet has been implemented with the idea of transforming the formalism of the Definition into SML source code as directly as possible. Not everything can be translated 1-to-1, though, because of the non-deterministic nature of some aspects of the rules, and also due to the set of additional informal rules that describe parts of the language.

Still, much care has been taken to get even the obscure details of these parts of the semantics right. For example, HaMLet goes to some length to treat the following correctly:

- not accepting additional syntactic phrases (e.g. with `as` or `fun`),
- parsing of the `where type` ... `and` derived form,
- expansion of derived forms (e.g. `withtype`, definitional type specifications),
- checking syntactic restrictions separately,
- `val rec` (binding rules, dynamic semantics),
- distinction of type variables from undetermined types,
- overloading resolution,
- flexible records,
- dynamic semantics.

The `test` directory in the HaMLet distribution contains some contrived examples exercising these corner cases and other code that is rejected by several SML systems despite being correct according to the Definition. HaMLet accepts all but two of them. Consequently, we are positive that HaMLet is more accurate in implementing the SML language specification than most other systems. There still are some deviations, though:

- inability to parse some legal SML programs (especially `fun`/`case`, see 4.4),
- non-principal types for equality polymorphic functions in `abstype` (see A.3),
- non-principal types for non-generalized declarations in functors (see A.4).

We consider all of these minor, since no existing SML implementations is able to deal with them. They are arguably mistakes on the side of the Definition, see A.8, A.1 and A.3. Still, we hope to fix these issues in future releases. Moreover, we plan to provide a more complete implementation of the Standard Basis Library.

# Acknowledgements

Thanks go to the following people who knowingly or unknowingly helped in putting together HaMLet and its documentation:

- Stefan Kahrs, Claudio Russo, Matthias Blume, Matthew Fluet, Derek Dreyer, Stephen Weeks, Bob Harper, Greg Morrisett, John Reppy, John Dias, David Matthews, Yan Chen, and people on the sml-implementers list for discussions about aspects and rough edges of the SML semantics,
- all people participating in the discussions on the sml-evolution list, the Successor ML wiki, and the SML evolution meeting,
- the authors of the original ML Kit [BRTT93], for their great work that inspired HaMLet,
- of course, the designers of ML and authors of the Definition, for the magnificent language. :)

# A Mistakes and Ambiguities in the Definition

This appendix lists all bugs, ambiguities and 'grey areas' in the Definition that are known to the author. Many of them were already present in the previous SML'90 version of the Definition [MTH90] (besides quite a lot that have been corrected in the revision) and are covered by Kahrs [K93, K96] in detail. Bugs new to SML'97 or not covered by Kahrs are marked with * and (*), respectively.

Where appropriate we give a short explanation and rationale of how we fixed or resolved an issue for HaMLet.

## A.1 Issues in Chapter 2 (Syntax of the Core)

Section 2.4 (Identifiers):

- The treatment of = as an identifier is extremely ad-hoc. The wording suggests that there are in fact two variants of the identifier class VId, one including and the other excluding = . The former is used in expressions, the latter everywhere else.

Section 2.5 (Lexical analysis):

- In [2.2] the Definition includes only space, tab, newline, and formfeed into the set of obligatory formatting characters that are allowed in source code. However, some major platforms require use of the carriage return character in text files. In order to achieve portability of sources across platforms it should be included as well.

  For consistency, HaMLet allows all formatting characters, for which there is explicit escape syntax, i.e. it includes vertical tab and carriage return.

Section 2.6 (Infixed Operators):

- The Definition says that "the only required use of `op` is in prefixing a non-infixed occurrence of an identifier which has infix status". This is rather vague, since it is not clear whether occurrences in constructor and exception bindings count as "non-infixed" [K93].

  We assume that `op` is only necessary in expressions and patterns and completely optional in constructor and exception bindings. This is consistent with the fact that `op` is not even allowed in the corresponding descriptions in signatures.

Section 2.8 (Grammar), Figure 4 (Expressions, Matches, Declarations and Bindings):

- (*) The syntax rules for $dec$ are highly ambiguous. The productions for empty declarations and sequencing allow the derivation of arbitrary sequences of empty declarations for any input.

  HaMLet does not allow empty declarations as part of sequences without a separating semicolon. On the other hand, every single semicolon is parsed as a sequence of two empty declarations.

- Another ambiguity is that a sequence of the form $dec_1\ dec_2\ dec_3$ can be reduced in two ways to $dec$: either via $dec_{12}\ dec_3$ or via $dec_1\ dec_{23}$ [K93]. See also A.2.

  We choose right associative sequencing, i.e. the latter parse, because that is most in line with the syntax for toplevel declarations.

Section 2.9 (Syntactic Restrictions):

- * The restriction that *valbind*s may not bind the same identifier twice (2nd bullet) is not a syntactic restriction as it depends on the identifier status of the *vid*s in the patterns of a *valbind*. Identifier status is derived by the elaboration rules. Similarly, the restriction on type variable shadowing (last bullet) is dependent on context and computation of unguarded type variables [Section 4.6].

  We implement checks for syntactic restrictions as a separate inference pass over the complete program that closely mirrors the static semantics. Ideally, all syntactic restrictions rather should have been defined as appropriate side conditions in the rules of the static *and* dynamic semantics by the Definition. Interestingly, semantic checks are already done for duplicate variables in patterns (rules 39 and 43), whereas these were still syntactic restrictions in the SML'90 edition.

- * An important syntactic restriction is missing:

  > "Any *tyvar* occurring on the right side of a *typbind* or *datbind* of the form *tyvarseq tycon* = $\cdots$ must occur in *tyvarseq*."

  This restriction is analogous to the one given for *tyvar*s in type specifications [3.5, item 4]. Without it the type system would be unsound. [12]

  We added a corresponding check.

## A.2 Issues in Chapter 3 (Syntax of Modules)

Section 3.4 (Grammar for Modules), Figure 6 (Structure and Signature Expressions):

- The syntax rules for *strdec* contain the same ambiguities with respect to sequencing and empty declarations as those for *dec* (see A.1).

  Consequently, we use equivalent disambiguation rules.

- Moreover, there are two different ways to reduce a sequence $dec_1$ $dec_2$ of core declarations into a *strdec*: via $strdec_1$ $strdec_2$ and via *dec* [K93]. Both parses are not equivalent since they provide different contexts for overloading resolution [Appendix E]. For example, appearing on structure level, the two declarations

  ```
  fun f x = x + x
  val a = f 1.0
  ```

  may be valid if parsed as *dec*, but do not type check if parsed as $strdec_1$ $strdec_2$ because overloading of + gets defaulted to `int`.

  We choose to always reduce to *strdec* as soon as possible, because that variant is simpler to implement and solves other problems as well (see A.7). Note that we use smaller contexts for overloading resolution (see 5.8) so that the way of parsing here actually would have no effect on the admissibility of programs.

- Similarly, it is possible to parse a structure-level `local` declaration containing only core declarations in two ways: as a *dec* or as a *strdec* [K93]. This produces the same semantic ambiguity.

  As above, we reduce to *strdec* as early as possible.

Section 3.4 (Grammar for Modules), Figure 7 (Specifications):

[12] Interestingly enough, in the SML'90 Definition the restriction was present, but the corresponding one for specifications was missing [MT91, K93].

- Similar as for *dec* and *strdec*, there exist ambiguities in parsing empty and sequenced *spec*s.

  We resolve them consistently.

- The ambiguity extends to sharing specifications. Consider:

  ```
  type t
  type u
  sharing type t = u
  ```

  This snippet can be parsed in at least three ways, with the sharing constraint taking scope over either both, or only one, or neither type specification. Since only the first alternative can be elaborated successfully, the validity of the program depends on how the ambiguity is resolved.

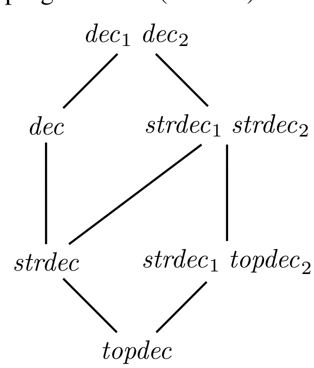
  We always extend the scope of a sharing constraint as far to the left as possible. That is a conservative choice, since all shared types must be specified in the respective scope and specifications may not contain duplicate type constructors.

Section 3.4 (Grammar for Modules), Figure 8 (Functors and Top-level Declarations):

- * Finally, another ambiguity exists for reducing a sequence $strdec_1\ \ strdec_2$ to a $topdec$: it can be done either by first reducing to $strdec$, or to $strdec_1\ topdec_2$. The latter is more restrictive with respect to free type variables (but see A.12 with regard to this).

  We use a consistent disambiguation method, i.e., reduce as early as possible.

Altogether, ignoring the infinite number of derivations involving empty declarations, the grammar in the Definition allows three ambiguous ways to reduce a sequence of two *dec*s to a *topdec*, as shown by the following diagram. All imply different semantics. The corresponding diagram for a sequence of three declarations would merely fit on a page. A further ambiguity arises at the program level (see A.7).




## A.3 Issues in Chapter 4 (Static Semantics for the Core)

Section 4.8 (Non-expansive Expressions):

- * The definition of non-expansiveness is purely syntactic and does only consider the right-hand side of a binding. However, an exception may result from matching against a non-exhaustive pattern on the left-hand side. It is rather inconsistent to disallow `raise` expressions in non-expansive bindings but allow implicit exceptions in the disguise of pattern match failure. More seriously, the possibility of exceptions stemming from polymorphic bindings is incompatible with type passing implementations.

This is no real bug but rather a design error. HaMLet implements the Defintion as is.

Section 4.9 (Type Structures and Type Environments):

- The definition of the Abs operator demands introduction of “new distinct” type names. However, type names can only be new relative to a context. To be precise, Abs would thus need an additional argument $C$ [K96].

  This is no issue on the implementation side, since fresh type names can simply be generated through stamping.

- Values in `abstype` declarations that are potentially polymorphic but require equality have no principal type [K96]. For example, in the declaration

  ```
  abstype t = T with
    fun eq(x, y) = x = y
  end
  ```

  the principal type of `eq` *inside* the scope of `abstype` clearly is `''a * ''a -> bool`. However, outside the scope this type is not principal because `''a` cannot be instantiated by `t`. Neither would `t * t -> bool` be principal, of course. Although not strictly a bug (there is nothing which enforces the presence of principal typings in the revised Definition), this semantics is very hard to implement faithfully, since type inference would have to deal with unresolved type schemes and to cascadingly defer decisions about instantiation and generalisation until the correct choice is determined.

  Like all other SML implementations, HaMLet assigns `eq` the type `''a * ''a -> bool`.

- A related problem is the fact that the rules for `abstype` may infer type structures that do not respect equality [K96]:

  ```
  abstype t = T with
    datatype u = U of t
  end
  ```

  Outside the scope of this `abstype` declaration type `u` will still be an equality type. Values of type `t` can thus be compared through the backdoor:

  ```
  fun eqT(x, y) = U x = U y
  ```

  HaMLet conforms to the behaviour implied by the Definition.

Section 4.10 (Inference Rules):

- * Rule 18 concerning datatype replication does not actually require the type to be a datatype. For example, the following is legal:

  ```
  datatype t = datatype unit
  ```

  The same applies to datatype replication in signatures (see A.4).

  Arguably, this is a design mistake, but we have no reason to change it for HaMLet.

- * The comment to rule 26 states that a declaration like

  ```
  datatype t = T
  val rec T = fn x => x
  ```

is legal since $C + VE$ overwrites identifier status. However, this comment overlooks an important point: in the corresponding rule 126 of the dynamic semantics recursion is handled differently, so that the identifier status is *not* overwritten. Consequently, the second declaration will raise a `Bind` exception. It clearly is an ill-design to infer inconsistent identifier status in the static and dynamic semantics, but fortunately it does not violate soundness in this case. Most implementations do not implement the 'correct' dynamic semantics, though.

HaMLet takes the specification litererally.

- * There is an unmatched left parenthesis in the consequent of rule 28.

Section 4.11 (Further Restrictions):

- (*) Under item 1 the Definition states that "the program context" must determine the exact type of flexible records, but it does not specify any bounds on the size of this context. Unlimited context is clearly infeasible since it is incompatible with `let` polymorphism: at the point of generalisation the structure of a type must be determined precisely enough to know what we have to quantify over.[13]

  In HaMLet, we thus restrict the context for resolving flexible records to the innermost surrounding value declaration, as most other SML systems seem to do as well. This is in par with our treatment of overloading (see 5.8).

  Note that some SML systems implement a slightly more restrictive variant, in which the following program does not type-check:

  ```
  fun f(r as {...}) =
      [let fun g() = r in r end, r : {a : int}]
  ```

  while a minor variation of it does:

  ```
  fun f(r as {...}) =
      [r : {a : int}, let fun g() = r in r end]
  ```

  The reason is that these implementations simply check for existence of unresolved record types in value environments to be closed, without taking into account that these types might stem from the context (in which case we know that we cannot quantify over the unknown bits anyway). As the above example shows, such an implementation compromises the compositionality of type inference. The Definition should rule it out somehow. A similar clarification is probably in order for overloading resolution (see A.11).

- Under item 2 the Definition demands that a compiler must give warnings whenever a pattern is redundant or a match is non-exhaustive. However, this requirement is inconsistent for two reasons:

  1. * There is no requirement for datatype constructors in sharing specifications or type realisations to be consistent. For example,

     ```
     datatype t = A | B
     datatype u = C
     sharing type t = u
     ```

     is a legal specification. Likewise,

     ```
     sig datatype t = A | B end where type t = bool
     ```

[13] Alternatively, there are extensions to Hindley/Milner typing that allow quantification over the structure of records, but polymorphic records are clearly not supported by the Definition.

is valid. Actually, this may be considered a serious bug on its own, although the Definition argues that inconsistent signatures are "not very significant in practice" [Section G.9]. If such an inconsistent signature is used to specify a functor argument, it allows a mix of constructors to appear in matches in the functor's body, rendering the terms of irredundancy and exhaustiveness completely meaningless.

There is no simple fix for this. HaMLet makes no attempt to detect this situation, so generation of warnings is arbitrary in this case.

2. (*) It is difficult in general to check equality of exception constructors – they may or may not be aliased. Inside a functor, constructor equality might depend on the actual argument structure the functor is applied to. It is possible to check all this by performing a certain amount of partial evaluation (such that redundant matches are detected at functor application), but this is clearly infeasible weighed against the benefits, in particular in conjunction with separate compilation.

   In HaMLet we only flag exception constructors as redundant when they are denoted by the same syntactic *longvid*. We do not try to derive additional aliasing information.

## A.4 Issues in Chapter 5 (Static Semantics for Modules)

Section 5.7 (Inference Rules):

- * As a pedantic note, the rules 64 and 78 use the notation $\{t_1 \mapsto \theta_1, \cdots, t_n \mapsto \theta_n\}$ to specify realisations. However, this notation is not defined anywhere in the Definition for infinite maps like realisations – [4.2] only introduces it for finite maps.

- * More seriously, both rules lack side conditions to ensure consistent arities for domain and range of the constructed realisation. Because $\varphi$ can hence fail to be well-formed [5.2], the application $\varphi(E)$ is not well-defined. The necessary side conditions are:

$$t \in \text{TyName}^{(k)} \tag{64}$$

$$t_i \in \text{TyName}^{(k)}, i = 1..n \tag{78}$$

  HaMLet adds the respective checks.

- * The presence of functors provides a form of explicit polymorphism which interferes with principal typing in the core language. Consider the following example [DB07]:

```
functor F(type t) =
  struct val id = (fn x => x) (fn x => x) end
structure A = F(type t = int)
structure B = F(type t = bool)
val a = A.id 3
val b = B.id true
```

  The declaration of `id` cannot be polymorphic, due to the value restriction. On the other hand, assigning it type `t -> t` would make the program valid. However, finding this type would require the type inference algorithm to skolemize all undetermined types in a functor body's result signature over the types appearing in its argument signature, and then perform a form of higher-order unification. Consequently, almost all existing implementations reject the program.[14]

HaMLet ignores this problem, rejecting the program due to a failure unifying types `int` and `bool`.

- * Just like in the core language (see A.3), rule 72 concerning datatype replication does not actually require the type to be a datatype.

- * The side conditions on free type variables in rules 87 and 89 do not have the effect that obviously was intended, see A.12 for details.

  HaMLet not only tests for free type variables, but also for undetermined types (see 5.6). This behaviour is not strictly conforming to the *formal* rules of the Definition (which define a more liberal regime), but meets the actual intention explicitly stated in [G.8]. It also is consistent with HaMLet's goal to always implement the most restrictive reading.

## A.5 Issues in Chapter 6 (Dynamic Semantics for the Core)

Section 6.4 (Basic Values):

- The APPLY function has no access to program state. This suggests that library primitives may not be stateful, implying that a lot of interesting primitives could not be added to the language without extending the Definition itself [K93].

  On the other hand, any non-trivial library type (e.g. arrays or I/O streams) requires extension of the definition of values or state anyway (and equality types – consider `array`). The Definition should probably contain a comment in this regard.

  HaMLet implements stateful library types by either mapping them to references in the target language (e.g. arrays) or by maintaining the necessary state outside the semantic objects (see 8.4).

## A.6 Issues in Chapter 7 (Dynamic Semantics for Modules)

Section 7.2 (Compound Objects):

- * In the definition of the operator $\downarrow$ : Env × Int → Env, the triple "$(SI, TE, VI)$" should read "$(SI, TI, VI)$".

Section 7.3 (Inference Rules):

- * Rule 182 contains a typo: both occurrences of $IB$ have to be replaced by $B$. The rule should actually read:

$$\frac{\mathrm{Inter}B \vdash \mathit{sigexp} \Rightarrow I \qquad \langle B \vdash \mathit{funbind} \Rightarrow F\rangle}{\begin{array}{r}B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{)}\ \texttt{=}\ \mathit{strexp}\ \langle\texttt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \{\mathit{funid} \mapsto (\mathit{strid} : I, \mathit{strexp}, B)\}\langle + F\rangle\end{array}} \tag{182}$$

[14] Interestingly, MLton [CFJW05] accepts the program, thanks to its defunctorization approach. However, it likewise accepts similar programs that are *not* valid Standard ML, e.g.:

```
functor F() = struct val id = (fn x => x) (fn x => x) end
structure A = F()
structure B = F()
val a = A.id 3
val b = B.id true
```

- * The rules for toplevel declarations are wrong: in the conclusions, the result right of the arrow must be $B'\langle+B''\rangle$ instead of $B'\langle'\rangle$ in all three rules:

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad B' = E \text{ in Basis} \qquad \langle B + B' \vdash \mathit{topdec} \Rightarrow B''\rangle}{B \vdash \mathit{strdec}\ \langle \mathit{topdec}\rangle \Rightarrow B'\langle+B''\rangle} \tag{184}$$

$$\frac{\text{Inter}B \vdash \mathit{sigdec} \Rightarrow G \qquad B' = G \text{ in Basis} \qquad \langle B + B' \vdash \mathit{topdec} \Rightarrow B''\rangle}{B \vdash \mathit{sigdec}\ \langle \mathit{topdec}\rangle \Rightarrow B'\langle+B''\rangle} \tag{185}$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F \qquad B' = F \text{ in Basis} \qquad \langle B + B' \vdash \mathit{topdec} \Rightarrow B''\rangle}{B \vdash \mathit{fundec}\ \langle \mathit{topdec}\rangle \Rightarrow B'\langle+B''\rangle} \tag{186}$$

## A.7 Issues in Chapter 8 (Programs)

- (*) The comment to rule 187 states that a failing elaboration has no effect. However, it is not clear what infix status is in scope after a failing elaboration of a program that contains top-level infix directives.

  HaMLet keeps the updated infix status.

- * There is another syntactic ambiguity for programs. A note in [3.4, Figure 8] restricts the parsing of *topdec*s:

  > "No *topdec* may contain, as an initial segment, a *strdec* followed by a semicolon."

  The intention obviously is to make parsing of toplevel semicolons unambiguous so that they always terminate a program. As a consequence of the parsing ambiguities for declaration sequences (see A.2) the rule is not sufficient, however: a sequence $\mathit{dec}_1$`;` $\mathit{dec}_2$`;` of core level declarations with a terminating semicolon can be first reduced to *dec*`;`, then to *strdec*`;`, and finally *program*. This derivation does not exhibit an "initial *strdec* followed by a semicolon." Consequently, this is a valid parse, which results in quite different behaviour with respect to program execution.

  Since HaMLet reduces to *strdec* as early as possible (see A.2), it works in the spirit of the Definition's intention.

- (*) The negative premise in rule 187 has unfortunate implications: interpreted strictly it precludes any conforming implementation from providing any sort of conservative semantic extension to the language. Any extension that allows declarations to elaborate that would be illegal according to the Definition (e.g. consider polymorphic records) can be observed through this rule and change the behaviour of consecutive declarations. Consider for example:

  ```
  val s = "no";
  strdec
  val s = "yes";
  print s;
  ```

  where the *strdec* only elaborates if some extension is supported. In that case the program will print `yes`, otherwise `no`.

  This probably indicates that formalising an interactive toplevel is not worth the trouble.

## A.8 Issues in Appendix A (Derived Forms)

Text:

- (*) The paragraph explaining rewriting of the *fvalbind* form rules out mixtures of *fvalbind*s and ordinary *valbind*s. However, the way it is formulated it does not rule out all combinations. It should rather say that all value bindings of the form $pat = exp$ `and` *fvalbind* or `rec` *fvalbind* are disallowed.

  HaMLet assumes this meaning.

Figure 15 (Derived forms of Expressions):

- The Definition is somewhat inaccurate about several of the derived forms of expressions and patterns. It does not make a proper distinction between atomic and non-atomic phrases. Some of the equivalent forms are not in the same syntactic class [MT91, K93].

  We assume the necessary parentheses in the desugared forms.

Figure 17 (Derived forms of Function-value Bindings and Declarations):

- The syntax of *fvalbind*s as given in the Definition enforces that all type annotations are *syntactically* equal, if given. This is unnecessarily restrictive and almost impossible to implement [K93], and probably not what was intended.

  HaMLet implements a more permissive syntax, as given by:

$$
\begin{array}{lllll}
 & \langle \texttt{op} \rangle vid\ atpat_{11} & \cdots\ atpat_{1n} & \langle : ty_1 \rangle & = exp_1 \\
\mid & \langle \texttt{op} \rangle vid\ atpat_{21} & \cdots\ atpat_{2n} & \langle : ty_2 \rangle & = exp_2 \\
\mid & \qquad \cdots & \quad \cdots & & \\
\mid & \langle \texttt{op} \rangle vid\ atpat_{m1} & \cdots\ atpat_{mn} & \langle : ty_m \rangle & = exp_m \\
 & & & \langle \texttt{and}\ fvalbind \rangle &
\end{array}
$$

  See also A.9 for a definition of the full syntax including infix notation.

Figure 19 (Derived forms of Specifications and Signature Expressions):

- * The derived form that allows several definitional type specifications to be connected via `and` is defined in a way that makes its scoping rules inconsistent with all other occurrences of `and` in the language. In the example

```
type t = int
signature S =
sig
  type t = bool
  and  u = t
end
```

  type `u` will be equal to `bool`, not `int` like in equivalent declarations. It would have been more consistent to rewrite the derived form to

```
include
  sig type tyvarseq_1 tycon_1
       and ···
       ···
       and tyvarseq_n tycon_n
  end where type tyvarseq_1 tycon_1 = ty_1
      ···
      where type tyvarseq_n tycon_n = ty_n
```

and delete the separate derived form for single definitional specifications.

This is a design error, but HaMLet implements it.

- * The Definition defines the phrase

  $$\mathit{spec}\ \texttt{sharing}\ \mathit{longstrid}_1 = \cdots = \mathit{longstrid}_n$$

  as a derived form. However, this form technically is not a derived form, since it cannot be rewritten in a purely syntactic manner – its expansion depends on the static environment.

  HaMLet thus treats this form as part of the bare grammar. Unfortunately, it is surprisingly difficult to formulate a proper inference rule describing the intended static semantics of structure sharing constraints – probably one of the reasons why it has been laxly defined as a derived form in the first place. The implementation simply collects all expanded type equations and calculates a suitable realisation incrementally. (At least there is no need for a corresponding rule for the dynamic semantics, since sharing qualifications are omitted at that point.)

- * The derived form for type realisations connected by `and` is not only redundant and alien to the rest of the language (`and` is nowhere else followed by a second reserved word), it also is extremely tedious to parse, since this part of the grammar is LALR(2) as it stands. It can be turned into LALR(1) only by a bunch of really heavy transformations. Consequently, almost no SML system seems to be implementing it correctly. Even worse, several systems implement it in a way that leads to rejection of programs *not* using the derived form. For example,

  ```
  signature A = S where type t = u where type v = w
  ```

  or

  ```
  signature A = S where type t = u
  and       B = T
  ```

  HaMLet goes to some length to do it correctly.

- * For complex type declarations the `withtype` derived form is important. With the introduction of equational type specifications in SML'97 it would have been natural to introduce an equivalent derived form for signatures. This is an oversight that most SML systems 'correct'.

  HaMLet stays with the language definition as is.

## A.9 Issues in Appendix B (Full Grammar)

Text:

- (*) To be pedantic, the first sentence is not quite true since there is a derived form for programs [Appendix A, Figure 18]. Moreover, it is not obvious why the appendix refrains from also providing a full version of the module and program grammar. It contains quite a lot of derived forms as well, and the section title leads the reader to expect it.

- The Definition gives precedence rules for disambiguating expressions, stating that "the use of precedence does not increase the class of admissible phrases". However, the rules are not sufficient to disambiguate all possible phrases. Moreover, for some phrases they actually rule out *any* possible parse, e.g.

```
a andalso if b then c else d orelse e
```

has no valid parse according to these rules. So the above statement is rather inconsistent [K93].

The HaMLet parser just uses Yacc precedence declarations for expression keywords that correspond to the precedence hierarchy given in the Definition. This seems to be the best way to approximate the intention of the Definition's rules.

- There is no comment on how to deal with the most annoying problem in the full grammar, the infinite look-ahead required to parse combinations of function clauses and `case` expressions, like in:

  ```
  fun f x = case e1 of z => e2
    | f y = e3
  ```

  According to the grammar this ought to be legal. However, parsing this would either require horrendous grammar transformations, backtracking, or some nasty and expensive lexer hack [K93]. Consequently, there is no SML implementation being able to parse the above fragment.

  HaMLet is no better in this regard.

Figure 21 (Grammar: Declarations and Bindings):

- The syntax given for *fvalbind* is incomplete, as is pointed out by the corresponding note. This is not really a bug but sloppy enough to cause some divergence among implementations.

  To make the grammar more precise we introduce the additional phrase classes *fmatch*, *fmrule*, and *fpat* and define them in analogy to *match*, *mrule*, and *pat*:

$$
\begin{array}{lcll}
\mathit{fvalbind} & ::= & \mathit{fmatch}\ \langle \texttt{and}\ \mathit{fvalbind}\rangle & \\
\mathit{fmatch} & ::= & \mathit{fmrule}\ \langle \texttt{|}\ \mathit{fmatch}\rangle & \\
\mathit{fmrule} & ::= & \mathit{fpat}\ \langle \texttt{:}\ \mathit{ty}\rangle\ \texttt{=}\ \mathit{exp} & \\
\mathit{fpat} & ::= & \langle \texttt{op}\rangle \mathit{vid}\ \mathit{atpat}_1\ \cdots\ \mathit{atpat}_n & (n \geq 1) \\
 & & \texttt{(}\mathit{atpat}_1\ \mathit{vid}\ \mathit{atpat}_2\texttt{)}\ \mathit{atpat}_3\ \cdots\ \mathit{atpat}_n & (n \geq 3) \\
 & & \mathit{atpat}_1\ \mathit{vid}\ \mathit{atpat}_2 &
\end{array}
$$

  This grammar is in accordance with our relaxation of type annotations in the *fvalbind* derived form (see A.8).

Figure 22 (Grammar: Patterns):

- While there are additional non-terminals *infexp* and *appexp* to disambiguate parsing of infix expressions, there is no such disambiguation for patterns. This implies that a pattern like `x:t ++ y` could be parsed if ++ was an appropriate infix constructor [K96]. Of course, this would result in heavy grammar conflicts.

  Since this appears to be an oversight, HaMLet does not allow such parsing. Constructor application always has higher precedence than type annotation. The full grammar of patterns thus is

$$
\begin{array}{lcl}
\mathit{atpat} & ::= & \text{...like before...} \\
\mathit{patrow} & ::= & \text{...like before...} \\
\mathit{apppat} & ::= & \mathit{atpat} \\
 & & \langle \texttt{op}\rangle \mathit{longvid}\ \mathit{atpat} \\
\mathit{infpat} & ::= & \mathit{apppat} \\
 & & \mathit{infpat}_1\ \mathit{vid}\ \mathit{infpat}_2 \\
\mathit{pat} & ::= & \mathit{infpat} \\
 & & \mathit{pat}\ \texttt{:}\ \mathit{ty} \\
 & & \langle \texttt{op}\rangle \mathit{vid}\ \langle \texttt{:}\ \mathit{ty}\rangle\ \texttt{as}\ \mathit{pat}
\end{array}
$$

with new phrase classes AppPat and InfPat. Similar to expressions, we get the following inclusion relation:

$$\text{AtPat} \subset \text{AppPat} \subset \text{InfPat} \subset \text{Pat}$$

Note that we actually do not need to distinguish between AppPat and InfPat, since there is no curried application in patterns. We do it nevertheless, for consistency.

## A.10 Issues in Appendix D (The Initial Dynamic Basis)

- (*) The Definition does specify the minimal initial basis but it does not specify what the initial state has to contain. Of course, it should at least contain the exception names `Match` and `Bind`.

  We define

  $$s_0 = (\{\}, \{\texttt{Match}, \texttt{Bind}\})$$

- The Definition does nowhere demand that the basis a library provides has to be consistent in any way. Nor does it require consistency between initial basis and initial state.

  The HaMLet library is consistent, of course.

## A.11 Issues in Appendix E (Overloading)

Overloading is the most hand-waving part of the otherwise pleasantly accurate Definition. Due to the lack of formalism and specific rules, overloading resolution does not work consistently among SML systems. For example, type-checking of the following declaration does not succeed on all systems:

```
fun f(x, y) = (x + y)/y
```

The existence of overloading destroys an important property of the language, namely the independence of static and dynamic semantics, as is assumed in the main body of the Definition. For example, the expressions

`2 * 100` and `2 * 100 : Int8.int`

will have very different dynamic behaviour, although they only differ in an added type annotation.

The Definition defines the overloading mechanism by enumerating all overloaded entities the library provides. This is rather unfortunate. It would be desirable if the rules were more generic, avoiding hard-coding overloading classes and the set of overloaded library identifiers on one hand, and allowing libraries to extend it in systematic ways on the other. More generic rules could also serve as a better guidance for implementing overloading (see 5.8 for a suitable approach).

The canonical way to deal with overloaded constants and value identifiers is to uniformingly assign an extended notion of type scheme that allows quantification to be constrained by an overloading class. Constraints would have to be verified at instantiation. This is more or less what has been implemented in HaMLet (see 5.8).

There are some more specific issues as well:

- * The Definition forgets to demand that any extension of a basic overloading class is consistent with respect to equality.

  Our formalisation includes such a restriction (see 5.8).

- * The Definition specifies an *upper* bound on the context a compiler may consider to resolve overloading, which is quite odd – of course, implementations cannot be prohibited to conservatively extend the language by making more programs elaborate. On the other hand, much more important would have been to specify a *lower* bound on what implementations *have to* support – it is clearly not feasible to force the programmer to annotate every individual occurence of an overloaded identifier or special constant.

  A natural and sensible lower bound seems to be the smallest enclosing core declaration that an overloaded identifier or constant appears in. We use that in HaMLet as the common denominator, consistent with the treatment of flexible records (see A.3).

Figure 27 (Overloaded Identifiers):

- * The types for the comparison operators <, >, <=, and >= must correctly be $\texttt{numtxt} \times \texttt{numtxt} \rightarrow \texttt{bool}$.

## A.12 Issues in Appendix G (What's New?)

Section G.8 (Principal Environments):

* At the end of the section the authors explain that the intent of the restrictions on free type variables at the toplevel (side-conditions in rules 87 and 89 [5.7]) is to avoid reporting free type variables to the user. However, judging from the rest of the paragraph, this reasoning confuses two notions of type variable: type variables as semantic objects, as appearing in the formal rules of the Definition, and the yet undetermined types during Hindley/Milner type inference, which are also typically represented by type variables. However, both kinds are variables on completely different levels: the former are part of the formal framework of the Definition, while the latter are an 'implementation aspect' that lies outside the scope of the Definition's formalism. Let us distinguish both by referring to the former as *semantic type variables* and to the latter as *undetermined types* (the HaMLet implementation makes the same distinction, in order to avoid exactly this confusion, see 5.2).

The primary purpose of the aforementioned restrictions obviously is to avoid reporting *undetermined types* to the user. However, they fail to achieve that. In fact, it is impossible to enforce such behaviour within the formal framework of the Definition, since it essentially would require formalising type inference (the current formalism has no notion of undetermined type). Consequently, the comment in Section [G.8] about the possibility of relaxing the restrictions by substituting arbitrary monotypes misses the point as well.

In fact, the formal rules of the Definition actually imply the exact opposite, namely that an implementation may *never* reject a program that results in undetermined types at the toplevel, and is thus compelled to report them. The reason is explicitly given in the same section: "implementations should not reject programs for which successful elaboration is possible". Consider the following program:

```
val r = ref nil;
r := [true];
```

Rule 2 has to non-deterministically choose some type $\tau$ `list` for the occurrence of `nil`. The choice of $\tau$ is not determined by the declaration itself: it is not used, nor can it be generalised, due to the value restriction. However, `bool` is a perfectly valid choice for $\tau$, and this choice will allow the entire program to elaborate. So according to the quote above, an implementation has to make exactly that choice. Now, if both declarations are entered separately into an interactive toplevel the implementation obviously has to defer commitment to that choice until it has actually seen the second declaration. Consequently, it can do nothing else but reporting an undetermined type for the first declaration. The only effect the side conditions in rules 87 and 89 have on this is that the types committed to later may not contain free semantic type variables – but considering the way such variables are introduced during type inference (mainly by generalisation), the only possibility for this is through a toplevel exception declaration containing a type variable (and such a declaration is indeed ruled out by those side conditions).[15]

There are two possibilities of dealing with this matter: (1) take the formal rules as they are and ignore the comment in the appendix, or (2) view the comment as an informal "further restriction" and fix its actual formulation to match the obvious intent. Since version 1.1.1 of HaMLet, we implement the intended meaning and disallow undetermined types on the toplevel, although this technically is a violation of the formal rules.

[15](*) Note that this observation gives rise to the question whether the claim about the existence of principal environments in Section 4.12 of the SML'90 Definition [MTH90] was valid in the first place. It most likely was not: a declaration like the one for `r` has no principal environment that would be expressible within the formalism of the Definition, despite allowing different choices of free imperative type variables. The reasoning that this relaxation was sufficient to regain principality is based on the same mix-up of semantic type variables and undetermined types as above. The relaxation does not solve the problem with expansive declarations, since semantic type variables are rather unrelated to it – choosing a semantic type variable for an undetermined type is no more principal than choosing any particular monotype.

# B History

## Version 1.0 (2001/10/04)

Public release. No history for prior versions.

## Version 1.0.1 (2001/10/11)

Basis:

- Fixed ASCII and Unicode escapes in `Char.scan` and `Char.scanC` (and thus in `Char.fromString`, `Char.fromCString`, `String.fromString`).
- Fixed octal escapes in `Char.toCString` (and thus `String.toCString`).
- Fixed possible NaN's in `Real.scan` for mantissa 0 and large exponents.

Documentation:

- Added issue of obligatory formatting characters to Appendix.
- Some minor additions/clarifications in Appendix.

Test cases:

- Added test case `redundant`.
- Removed accidental carriage returns from `asterisk`, `semicolon` and `typespec`.
- Small additions to `semicolon` and `valrec`.

## Version 1.1 (2002/07/26)

Basis:

- Adapted signatures to latest version of the Basis specification [GR04].
- Implemented new library functions and adapted functions with changed semantics.
- Implemented all signatures and structures dealing with array and vector slices.
- Implemented new `Text` structure, along with missing `CharVector` and `CharArray` structures.
- Implemented missing `Byte` structure.
- Removed `SML90` structure and signature.
- Use opaque signature constraints where the specification uses them (with some necessary exceptions).
- Implemented missing `Bool.scan` and `Bool.fromString`.
- Implemented missing `Real.posInf` and `Real.negInf`.
- Handle exceptions from `Char.chr` correctly.
- Fixed generation of `\^X`-escapes in `Char.toString`.
- Fixed treatment of gap escapes in `Char.scan`.

Test cases:

- Added test case `replication`.
- Updated conformance table.

## Version 1.1.1 (2004/04/17)

Interpreter:

- Disallow undetermined types (a.k.a. "free type variables") on toplevel.
- Implement accurate scope checking for type names.
- Fixed soundness bug w.r.t. undetermined types in type scheme generalisation test.
- Reject out-of-range real constants.
- Accept multiple line input.
- Output file name and line/columns with error messages.
- Improved pretty printing.

Basis:

- Sync'ed with updates to the specification [GR04]: overloaded $\sim$ on words, added `Word.fromLarge`, `Word.toLarge`, `Word.toLargeX`; removed `Substring.all`; changed `TextIO.inputLine`; changed `Byte.unpackString` and `Byte.unpackStringVec`.
- Fixed `String.isSubstring`, `String.fields`, and `Vector.foldri`.

Test cases:

- Added test cases `abstype2`, `dec-strdec`, `flexrecord2`, `tyname`, `undetermined2`, `undetermined3`.
- Split conformance table into different classes of deviation and updated it.

## Version 1.1.2 (2005/01/14)

Interpreter:

- Fix parsing of sequential and sharing specifications.
- Add arity checks missing in rules 64 and 78 of the Definition.
- Implement type name equality attribute as `bool`.

Basis:

- Fixed `StringCvt.padLeft` and `StringCvt.padRight`.

Documentation:

- Add parsing ambiguity for sharing specifications to issue list.
- Add missing side conditions in rules 64 and 78 to issue list.
- Added version history to appendix.

Test cases:

- Added test cases `poly-exception`, `tyvar-shadowing`, and `where2` and extended `id` and `valrec`.
- Updated conformance table.

## Version 1.2 (2005/02/04)

Interpreter:

- Refactored code: semantic objects are now collected in one structure for each part of the semantics; type variable scoping and closure computation (expansiveness check) are separated from elaboration module.
- Made checking of syntactic restrictions a separate inference pass.
- Added missing check for bound variables in signature realisation.
- Fixed precedence of environments for `open` declarations.
- Fixed implementation of Abs operator for `abstype`.
- Print type name set $T$ of inferred basis in elaboration mode.

- Fixed parenthesisation in pretty printing type applications.

Basis:

- More correct path resolution for `use` function.
- Added `checkFloat` to `REAL` signature so that bootstrapping actually works again.
- Fixed `ArraySlice.copy` for overlapping ranges.
- Fixed `ArraySlice.foldr` and `ArraySlice.foldri`.
- Fixed `Char.isSpace`.
- Fixed octal escapes in `Char.fromCString`.
- Updated treatment of trailing gap escapes in `Char.scan`.
- Updated scanning of hex prefix in `Word.scan`.
- Fixed traversal order in `Vector.map`.

Documentation:

- Added typo in rule 28 to issue list.

Test files:

- Added `generalise`.
- Extended `poly-exception`.

## Version 1.2.1 (2005/07/27)

Interpreter:

- Fixed bug in implementation of rule 35.
- Fixed bug in check for redundant match rules.

Basis:

- Fixed `Substring.splitr`.
- Fixed border cases in `OS.Path.toString`, `OS.Path.joinBaseExt`, `OS.Path.mkAbsolute`, and `OS.Path.mkRelative`.

## Version 1.2.2 (2005/12/09)

Interpreter:

- Simplified implementation of pattern checker.

Test files:

- Added `fun-infix`.

## Version 1.2.3 (2006/07/18)

Interpreter:

- Fixed check for duplicate variables in records and layered patterns.
- Added missing check for undetermined types in functor declarations.
- Overhaul of line/column computation and management of source file names.

Documentation:

- Added principal typing problem with functors to issue list.

Test files:

- Added `fun-partial`, `functor-poly` and `functor-poly2`.
- Updated conformance table.

### Version 1.2.4 (2006/08/14)

Documentation:

- Clarified license.

### Version 1.3.0 (2007/03/22)

Interpreter:

- Output abstract syntax tree in parsing mode.
- Output type and signature environments in evaluation mode.
- Fixed computation of tynames on a static basis.
- Reorganised directory structure.
- Some clean-ups.

Documentation:

- Updated a few out-of-sync sections.
- Added typo in definition of $\downarrow$ operator (Section 7.2) to issues list.

Test files:

- Extended `sharing` and `where`.
- Updated conformance table.

Platforms:

- Support for Poly/ML, Alice ML, and the ML Kit.
- Support for incremental batch compilation with Moscow ML and Alice ML.
- Target to build a generic monolithic source file.

### Version 1.3.1 (2008/04/28)

Platforms:

- Preliminary support for SML#.
- Avoid name clash with library of SML/NJ 110.67.
- Avoid shell-specific code in `Makefile`.

### Version 2.0.0 (2013/10/10)

Interpreter functionality:

- Print source location for uncaught exceptions.
- Abort on errors in batch modes.
- New command line option -b to switch standard basis path (or omit it).
- Fixed bug in lexing of negative hex constants (thanks to Matthew Fluet).
- Fixed missing identifier status check for variable in ‘as’ patterns.

- Fixed missing type arity check for structure sharing derived form.
- Slightly more faithful checking of syntactic restrictions (ignore duplicate variables in matches).
- Slightly more faithful handling of equality maximisation (don't substitute).
- Slightly more faithful handling of sharing specifications (don't generate new type names).

Interpreter implementation:

- Restructured AST to include annotations in the form of typed property list (breaks all code based on HaMLet 1, sorry :( ).
- Elaboration stores result of each rule as annotation in respective AST node.
- Derived forms make sure to clone nodes where necessary.
- Removed ad-hoc type annotation on SCons.
- Split Library into StaticLibrary and DynamicLibrary, to support compilers.
- Provide separate Elab/EvalProgram modules, to support compilers/interpreter.
- Renamed *Grammar* structures to *Syntax*.
- Tons of code clean-up and beautification.

JavaScript compiler:

- New HaMLet mode -j, "compile to JavaScript".
- Simple type-aware source-to-source translation into JavaScript.
- JavaScript implementation of Standard Basis Library primitives.

Basis:

- Implemented CommandLine.
- Skeletal implementation of OS.Process.
- Implemented Substring.position,tokens,fields.

Platforms:

- Assume Moscow ML 2.10 by default.
- Added workarounds for String.concatWith and CharVector.all for (old versions of) Moscow ML.
- Renamed hamlet-monolith.sml to hamlet-bundle.sml.

Documentation:

- Updated manual and man page.
- Added lax datatype replication rules to issue list (suggested by Karl Crary).
- Updated links.

## Version 2.0.1 (2025/07/27)

Interpreter:

- Fixed bug in evaluation order of 'open' with multiple structures (reported by Arata Mizuki).
- Fixed bug in treatment of equality attribute for ref type (thanks to El Pin Al).

Building:

- Use 'polyc' command for more reliable build with Poly/ML (thanks to Brian Campbell).
- Avoid backslashes in echo command, problematic on MacOS (thanks to Arata Mizuki).

# References


[MTHM97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen
*The Definition of Standard ML* (Revised)
The MIT Press, 1997

[MTH90] Robin Milner, Mads Tofte, Robert Harper
*The Definition of Standard ML*
The MIT Press, 1990

[MT91] Robin Milner, Mads Tofte
*Commentary on Standard ML*
The MIT Press, 1991

[K93] Stefan Kahrs
*Mistakes and Ambiguities in the Definition of Standard ML*
University of Edinburgh, 1993
`http://www.cs.ukc.ac.uk/pubs/1993/569/`

[K96] Stefan Kahrs
*Mistakes and Ambiguities in the Definition of Standard ML – Addenda*
University of Edinburgh, 1996
`ftp://ftp.dcs.ed.ac.uk/pub/smk/SML/errors-new.ps.Z`

[DB07] Derek Dreyer, Matthias Blume
*Principal Type Schemes for Modular Programs*
in: Proc. of the 2007 European Symposium on Programming
Springer-Verlag, 2007

[GR96] Emden Gansner, John Reppy
*The Standard ML Basis Library* (preliminary version 1996)
AT&T and Lucent Technologies, 2004
`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/`

[GR04] Emden Gansner, John Reppy
*The Standard ML Basis Library*
Cambridge University Press, 2004
`http://www.standardml.org/Basis/`

[DM82] Luis Damas, Robin Milner
*Principal type schemes for functional programs*
in: Proc. of 9th Annual Symposium on Principles of Programming Languages
ACM Press, 1982

[C87] Luca Cardelli
*Basic Polymorphic Typechecking*
in: *Science of Computer Programming* 8(2)
Elsevier Science Publisher, 1987

[S96] Peter Sestoft
*ML pattern match compilation and partial evaluation*
in: Dagstuhl Seminar on Partial Evaluation, LNCS 1110
Springer-Verlag 1996
`ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Sestoft/papers/match.ps.gz`

[W98] Philip Wadler
*A prettier printer*

in: The Fun of Programming
Palgrave Macmillan, 2003
`http://cm.bell-labs.com/cm/cs/who/wadler/`

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, David Turner
*The ML Kit* (Version 1)
`http://www.diku.dk/research-groups/topps/activities/kit2/mlkit1.html`

[K06] *The ML Kit*
`http://www.it-c.dk/research/mlkit/`

[NJ07] *Standard ML of New Jersey*
`http://www.smlnj.org/`

[NJ98] *The SML/NJ Library*
`http://www.smlnj.org/doc/smlnj-lib/`

[CFJW05] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, Stephen Weeks
*MLton User Guide*
`http://www.mlton.org/`

[M07] David Matthews
*Poly/ML*
`http://www.polyml.org/`

[RRS00] Sergei Romanenko, Claudio Russo, Peter Sestoft
*Moscow ML Owner's Manual* (Version 2.10)
`http://mosml.org`

[AT06] *The Alice Programming System*
`http://www.ps.uni-sb.de/alice/`

[ST07] *SML# Project*
`http://www.pllab.riec.tohoku.ac.jp/smlsharp/`

[TA00] David Tarditi, Andrew Appel
*ML-Yacc User Manual* (Version 2.4)
`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Yacc/manual.html`

[AMT94] Andrew Appel, James Mattson, David Tarditi
*A lexical analyzer generator for Standard ML* (Version 1.6.0)
`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Lex/manual.html`

[ES12] Ecma International
*ECMAScript Language Specification* (Edition 5.1)
`http://www.ecmascript.org`

[E08] Martin Elsman
*SMLtoJs*
`http://www.smlserver.org/smltojs/`