HaMLet S

To Become Or Not To Become Successor ML :-)

Version 1.3.1/S5
2008/04/28

Andreas Rossberg
Universität des Saarlandes
rossberg@ps.uni-sb.de

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. 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".

This special version of HaMLet is devoted to *Successor ML* [SML05], an envisioned evolutionary, conservative successor to Standard ML. It incorporates a number of preliminary proposals made for Successor ML and represents a personal vision of where SML could go. Currently, it concentrates on the following features:

- Extensible records.
- More expressive pattern matching.
- Views.
- Higher-order modules and nested signatures.
- Local and first-class modules.
- Miscellaneous fixes to known issues with SML and its specification.

See Appendix B for a detailed description of all changes relative to Standard ML.

1.1 Goals

The primary purpose of HaMLet is not to provide yet another SML system. Its goal is to provide 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 9 for an overview.

Some time ago, a loose for evolving SML has been started. For political reasons, the subject of this effort has been nicknamed *Successor ML* (sML) [SML05]. This special version of HaMLet is a testbed for potential changes and extensions considered for Successor ML and incorporates a number of simple proposals. Appendix B gives a complete list of these proposals and their specification.

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. Comfort was no priority either. The error messages given by HaMLet are usually taciturn as we tried to avoid complicating the implementation.

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.

Most of the problems have been fixed in this version as part of the proposals for Successor ML, see especially Appendices B.1 and B.2. The general approach we take for resolving remaining ambiguities and fixing bugs is doing it in the 'most natural' way. Mostly, this is obvious, sometimes it is not. The appendix discusses the solutions 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 is suitable to fill the resulting gap.

We also believe that HaMLet is considerably simpler and closer to the Definition. Moreover, unlike the ML Kit, it also 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

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@ps.uni-sb.de
```

2 Usage

2.1 Download

HaMLet is available from the following web page:

```
http://www.ps.uni-sb.de/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)[1] [K06]
- SML# (0.20 or higher)[2] [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 Libraries and Tools Used

HaMLet makes use of the Standard ML Basis Library [GR04][3]. 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.[4]

2.4 Installation

To build a stand-alone HaMLet program, go to the HaMLet source directory and invoke one of the following commands:[5]

```
make with-smlnj
make with-mlton
```

[1] Unfortunately, the ML Kit seems to hang itself compiling the parser module of the current version of HaMLet-S.

[2] Hamlet on SML# currently works with some glitches only, e.g. the interactive prompt does appear out of sync.

[3] 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].

[4] 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.

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

```
make with-poly
make with-mosml
make with-alice
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.[6]

The above `make` targets use the fastest method to build 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:[7]

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

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

```
make hamlet-monolith.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].[8]

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

[6] Due to a bug in Moscow ML, which does not parse SML's `where type` syntax correctly, you first have to run "`make mosmlize`" to patch the sources appropriately. Unfortunately, Hamlet will no longer be able to bootstrap from the patched sources, due to the language change described in Appendix B.7. By running "`make unmosmlize`" you can convert the sources back to their original form.

[7] 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).

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

- `-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)

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 minor glitch with regard to overloaded constants: since no type information is available in evaluation mode, all constants will be assigned the 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). You can still force calculation to be performed in 8 bit words by performing explicit conversions:

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

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

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 still is weak. Most basic types and corresponding operations are fully implemented, though.

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

  ```
  CM.make();
  ```

  However, under newer versions of SML/NJ (110.20 and later), you need to invoke the function as follows:

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

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

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

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

    val parseSession : unit -> unit
    val elabSession :  unit -> unit
```

```
    val evalSession :  unit -> unit
    val execSession :  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 four 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.

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.

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[9] (see Section 6).

[9] For example, on a 2 GHz processor with 512 MB memory the second iteration may take about 4 hours.

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 of the issues remaining with Successor ML:

- Exhaustiveness of Patterns: checking of patterns is not fully accurate in the presence of overloaded special constants. Sometimes a match is flagged as non-exhaustive, although it is in the limited range of its actual type.
- 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, ammended by the changes given in Appendix B, as closely as possible. The idea was 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 this 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. The text assumes that you have both at hand 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.

Note that most explanations given here a kept 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 implementing that aspect of the language, namely these are:

Chapter 2 and 3	`Lexer`, `Parser`, `SyntacticRestrictions`
Chapter 4	`ElabCore`
Chapter 5	`ElabModule`
Chapter 6	`EvalCore`
Chapter 7	`EvalModule`
Chapter 8	`Program`
Appendix A	`DerivedForms`
Appendix B	`Parser`
Appendix C	`InitialStaticBasis`
Appendix D	`InitialDynamicBasis`
Appendix E	`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 parts or entities of 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 of those sections 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*, `Grammars`*X*, 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.

We tried to keep things simple, so the architecture of HaMLet is quite flat: it does not make heavy use of functors. Functors only appear where the need to generate several instances of an abstract type (e.g. `IdFn`) or parameterised types arises. Enthusiasts of the closed functor style may feel free to dislike this approach `;-)`.

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 (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. To allow the most direct implementation of rules operating on semantic objects, type definitions representing structured sets are always kept transparent. Be warned: 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

$$A \vdash phrase \Rightarrow A'$$

the corresponding function has type

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

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

$$\frac{A_1 \vdash phrase_1 \Rightarrow A_1' \qquad \cdots \qquad A_n \vdash phrase_n \Rightarrow A_n' \qquad \text{side condition}}{A \vdash phrase \Rightarrow A'} \qquad (k)$$

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 side condition then
        A'
    else
        error("message")
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. At the following places state comes into play:

- inside the lexer, to handle nested comments,
- inside the parser, to maintain the infix environment,
- 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,
- to realise inter-module recursion on one occasion (see Section 5.11).

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.

3.7 Module-level Mutual Recursion

The addition of various module extensions (see Appendices B.23–B.26) introduces between the implementation of the core and the module language. Since SML does not support recursive modules, we either have to merge many conceptually separate concepts into a single module, or work around it. We chose the latter, using what can best be considered a hack:

- To break up inter-module type recursion, we abuse the exception type. In one structure, a the proper type is replaced by `exn`, while the other structure defines the actual type and an appropriate exception constructor wrapping it.
- On the value level inter-module recursion is always between functions. We use references to tie the recursive knot. One structure defines a reference as a placeholder for the actual function, and all calls are performed through the reference. The corresponding structure defining the proper function assigns this reference.

4 Abstract Syntax and Parsing

4.1 Files

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

`Source`	representation of source regions
`IdFn`	generic identifier representation
`LongIdFn`	
`IdsCore`	instantiated identifier classes
`IdsModule`	
`TyVar`	type variable representation
`Lab`	label representation
`SCon`	special constants
`GrammarCoreFn`	abstract syntax tree definition
`GrammarModuleFn`	
`GrammarProgramFn`	
`Grammars`	AST instantiations
`Lexer`	lexical analysis (via ML-Lex)
`LineAwareLexer`	wrapper computing line/column information
`Parser`	syntactical analysis (via ML-Yacc)
`Infix`	infix parser
`Parse`	parser plugging

`DerivedFormsCore`	derived forms transformation
`DerivedFormsModule`	
`DerivedFormsProgram`	
`IdStatus`	identifier status
`BindingObjectsCore`	objects for binding analysis
`BindingObjectsModule`	
`GenericEnvFn`	generic environment operations
`BindingEnv`	operations on binding environment
`BindingContext`	operations on binding context
`BindingBasis`	operations on binding basis
`ScopeTyVars`	scoping analysis for type variables
`SyntacticRestrictionsCore`	verifying syntactic restrictions
`SyntacticRestrictionsModule`	
`SyntacticRestrictionsProgram`	
`PPGrammar`	auxiliary functions for printing ASTs
`PPCore`	printing of core AST
`PPModule`	printing of module AST
`PPProgram`	printing of program AST

4.2 Abstract Syntax Tree

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 stuff has been removed from the core grammar, as it does not appear in the semantic rules of the Definition [2.6]. However, we have to keep occurrences of the `op` keyword in order to do infix resolution (see 4.5).

Each node carries a generic info field, and the grammar modules are functorised to allow different instantiations of this field. However, they are currently only instantiated once, with the info field carrying position information mapping each node to a region of the source text and an optional file name (file `Grammars`).

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 essential part of their lexical appearance – their actual values cannot be calculated before overloading resolution.

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 grammar types described in Section 4.2.

Most parts of the parser and lexer specifications (files `Parser.grm` and `Lexer.lex`) are straightforward. In particular, we use a rather dumb and direct way to recognize keywords in the lexer. 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

The SML grammar – even with the changes given in Appendix B.1 – contains several other ambiguities on the declaration level (see A.1, A.2 and A.7). We resolve them in the 'most natural' ways. In particular, semicolons are simply parsed as declarations or specifications, not as separators (cf. A.1), and several auxiliary phrase classes have been introduced to implement these disambiguations. Further grammar transformations are needed to cope with 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.

The Definition describes the $fvalbind$ derived form in a very inaccurate way. The change described in Appendix B.1 makes 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 has been 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:

`StaticObjectsCore`	definition of semantic objects
`StaticObjectsModule`	
`TyVar`	type variables
`TyName`	type names
`Type`	operations on types
`TypeFcn`	operations on type functions
`TypeScheme`	operations on type schemes
`OverloadingClass`	overloading classes
`GenericEnvFn`	generic environment operations
`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 3.3 (module `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. We only need simple kinds of additional arguments: 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 (module `ElabCore`).

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.

Similarly, the time stamp mechanism is used to prevent invalid unification of monomorphic undetermined types remaining due to the value restriction, with type variables, see Section 5.7.

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 τ'.

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 τ may contain undetermined types, stemming from expansive declarations. 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.[10] 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 try 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
```

[10] 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`.[11]

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.[12]

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

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

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 (additional argumetn `utaus`). 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 have to be annotated with corresponding type names by overloading resolution, in order to get the correct dynamic semantics (see 6.3) and enable proper checking of match exhaustiveness (see 5.12). For this purpose, the list of unresolved types can carry optional associated special constants. During defaulting we annotate each constant, and do range checking of the constant's value with respect to the resolved type at the same time.

5.9 Recursive Bindings and Datatype Declarations

Value bindings with `rec` and datatype declarations are recursive. The inference rules (15, 17 and 19 after the change from Appendix B.4) 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.

[11] Some SML implementations do not handle this case properly.

[12] 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.

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 type names introduced and later adjusting all invalid equality attributes in a fixpoint iteration until all type structures respect equality (function `StaticEnv.maximiseEquality`).

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 a module E' against a signature $\Sigma = (T, E)$ we first calculate an appropriate realisation φ 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 φ 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 φ. We can apply φ 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.

So far, the description applies to modules as defined in the Definition. The change in appendix B.23 generalises matching to higher-order modules. That means that modules M may appear instead of environments E in the above cases. Computing a realisation for matching is not complicated, though, since functors cannot bind any type names, so that T remains empty for functor signatures and only the case $M = E$ has to be considered, as before.

5.12 Checking Patterns

Section [4.11], items 2 and 3 require checking exhaustiveness and irredundancy of patterns. The algorithm for performing this check is based on [S96] (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 equalised (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:

`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
`GenericEnvFn`	generic environment operations

`DynamicEnv`	operations on environments
`Inter`	operations on interfaces
`DynamicBasis`	operations on basis
`EvalCore`	implementation of evaluation rules
`EvalModule`	

6.2 Value Representation

Values are represented as defined in Section 6.3 of the Definition (module `Val`). 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 `BasVal`). 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 only handles `=`. For all other basic values it dispatches to the `Library` 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 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

$$s, A \vdash phrase \Rightarrow A'/p, s'$$

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 elaboration 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).

7 Toplevel

7.1 Files

The remaining modules implement program execution and interactive toplevel:

`Basis`	the combined basis
`Program`	implementation of rules for programs
`InitialInfixEnv`	initial environments
`InitialStaticEnv`	
`InitialStaticBasis`	
`InitialDynamicEnv`	
`InitialDynamicBasis`	
`PrettyPrint`	pretty printing engine
`PPMisc`	auxiliary pretty printing functions
`PPType`	pretty printing of types
`PPVal`	... values
`PPStaticEnv`	... static environment
`PPStaticBasis`	... static basis
`PPDynamicEnv`	... dynamic environment
`PPDynamicBasis`	... dynamic basis
`PPBasis`	... combined basis
`Use`	the `use` queue
`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, an `Error` exception will 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).

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:

`Library`	primitive part of the library
`Use`	`use` queue
`basis/`	the actual library modules

8.2 Language/Library Interaction

The Definition contains several hooks where it explicitly delegates fleshing out stuff 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).

We encapsulate all library extensions into one single module `Library` that defines the parts of these objects that are left open by the Definition. However, we split up implementation of the library into two layers:

- the *primitive* layer that contains everything that cannot be defined within the target language,
- the *surface* layer which defines the actual library.

By *target language* we mean the language to be implemented. Many library entities are definable within the target 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 `Library` module defines everything that has to be primitive (see 8.3), while the rest is implemented within the target language in the modules inside the `basis` directory (see 8.6). These modules have to make assumptions about what is defined by the `Library` module, so that both actually 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.[13] 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`.

Everything else can be obtained from these in the target language. Primitive exceptions not available on the toplevel are wrapped into their residuent structures.

To enable the target language to bind the basic values defined by the library, we piggy-back the `use` function. Its dynamic semantics is overloaded and in the static basis exported by the `Library` module it is given type $\alpha \to \beta$. Applying it to a record of type $\{\texttt{b} : \texttt{string}\}$ will return the basic value denoted by the string `b` – of course, the library source code should annotate the result type properly to be type-safe. Primitive constants of type τ are available as functions $\texttt{unit} \to \tau$.

The `use` function has been chosen for this purpose since its existence cannot be encapsulated in the library anyway – the interpreter has to know about it (see 8.5). Once all necessary basic values have been bound, the library source code should hide the additional, unsafe functionality of `use` by rebinding it with its properly restricted type $\texttt{string} \to \texttt{unit}$.

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).

[13] 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.

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 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 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:

- checking syntactic restrictions separately,
- derived forms (e.g. `withtype`, definitional type specifications),
- distinction of type variables from undetermined types,
- overloading resolution,
- flexible records,
- dynamic semantics.

Some more issues present in SML'97 have been removed by the changes described in Appendix B.

Acknowledgements

Thanks go to the following people who knowingly or unknowingly helped in putting together HaMLet and this special "Successor ML" version:

- Stefan Kahrs, Claudio Russo, Matthias Blume, Derek Dreyer, Stephen Weeks, Bob Harper, Greg Morrisett, John Reppy, John Dias, David Matthews, and other 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 initially inspired the work on 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 it in 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.

  Fixed by change described in Appendix B.1.

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].

  Fixed by change described in Appendix B.1.

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. This makes parsing of empty declarations unambiguous.

- 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 left associative sequencing, i.e. the former parse.

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 vids in the patterns of a $valbind$. Identifier status can be derived by inference rules only. 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.

- * 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. [14]

  Fixed by change described in Appendix B.2.

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`.

  Fixed by change described in Appendix B.1.

- 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.

  Fixed by change described in Appendix B.1.

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

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

  We resolve them consistently.

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

- 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 ambiguity is resolved.

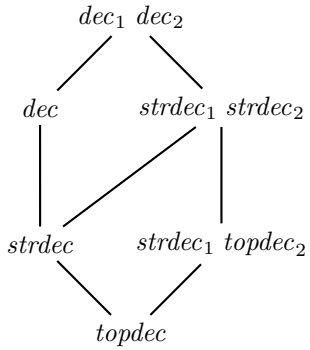
  Fixed by change described in Appendix B.1.

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).

  Fixed by change described in Appendix B.1.

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 decs 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).



All parsing ambiguities (except for ones involving empty declarations, which are harmless) are fixed by the changes described in Appendix B.1.

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.

  Fixed by change described in Appendix B.3.

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].

  Avoided by the change making `abstype` a derived form, as described in Appendix B.5.

- Values in `abstype` declarations that are potentially polymorphic but require equality types 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.

  Avoided by the change making `abstype` a derived form, as described in Appendix B.5. Abstract types no longer hide equality.

- 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
  ```

  Avoided by the change making `abstype` a derived form, as described in Appendix B.5. Abstract types no longer hide equality.

Section 4.10 (Inference Rules):

- * 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 omits 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 arguably is a serious 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.

  Removed by the change described in Appendix B.4.

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

  Fixed by change described in Appendix B.2.

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.[15]

  Fixed by change described in Appendix B.2.

  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 they 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 consistency of datatype constructors in sharing specifications or type realisations. 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
     ```

     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 completely arbitrary in this case.

[15] 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.

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 abstract interpretation (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):

- * 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.

  This is just a minor oversight, the intended meaning is obvious.

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

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

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

  Fixed by change described in Appendix B.2.

- * 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. Nevertheless, 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.[16]

[16] 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
```

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

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

  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] and 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 $\times$ Int $\to$ Env, the triple "(SI, TE, VI)" should read "(SI, TI, VI)".

  Fixed by change given in Appendix B.2.

Section 7.3 (Inference Rules):

- * Rule 182 contains a typo: both occurrences of IB have to be replaced by B.

  Fixed by change described in Appendix B.2.

- * 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.

  Fixed by change described in Appendix B.2.

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 dec_1`;` 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.

  Fixed by change described in Appendix B.1.

- (*) 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 equivalent 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].

  Fixed by change described in Appendix B.1.

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 occurences 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.

  Made consistent with the rest of the language by change described in Appendix B.6.

- * The Definition defines the phrase

  $$spec \texttt{ sharing } longstrid_1 \texttt{ = } \cdots \texttt{ = } longstrid_n$$

  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 completely 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
```

  Removed by change described in Appendix B.7.

- * 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'.

  Added by the extension described in Appendix B.22.

A.9 Issues in Appendix B (Full Grammar)

Text:

- (*) The first sentence is not 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.

  First issue fixed by change described in Appendix B.1.

- 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].

  Fixed by change described in Appendix B.1.

- 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.

  Ruled out by change described in Appendix B.1.

Figure 21 (Grammar: Declarations and Bindings):

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

  Fixed by change described in Appendix B.1.

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` can be parsed if `++` is an appropriate infix constructor [K96]. Of course, this would result in heavy grammar conflicts.

  Disambiguated by change described in Appendix B.1.

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`.

  Fixed by change described in Appendix B.2.

- 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 would be a bit more generic, avoiding hardcoding 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 for a suitable approach).

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.

  Fixed by change described in Appendix B.2.

  Our formalisation includes such a restriction (see 5.8).

- * That the Definition specifies an *upper* bound on the context a compiler may consider to resolve overloading 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.

Fixed by change described in Appendix B.2.

Figure 27 (Overloaded Identifiers):

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

  Fixed by change described in Appendix B.2.

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 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 τ `list` for the occurrence of `nil`. The choice of τ 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 τ, 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.[17]

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.

B Language Changes

In this appendix we describe all modifications and extensions to the Definition that are implemented in this version of HaMLet. Most of them have already been proposed for Successor ML and are taken from the discussion Wiki [SML05]. These can be put in two groups:

Fixes and simplifications:

- Syntax fixes
- Semantic fixes
- Monomorphic non-exhaustive bindings
- Simplified recursive value bindings
- Abstype as derived form
- Fixed manifest type specifications
- Abolish sequenced type realisations

Extensions:

- Line comments
- Extended literal syntax
- Record punning
- Record extension
- Record update
- Conjunctive patterns
- Disjunctive patterns
- Nested matches
- Pattern guards
- Transformation patterns

[17](*) 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 of `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.

- Optional bars and semicolons
- Optional else branch
- Views
- Do declarations
- Withtype in signatures
- Higher-order functors
- Nested signatures
- Local modules
- First-class modules

Examples demonstrating some of the more involved extensions in detail can be found in the `doc/examples` directory of the distribution.

B.1 Syntax Fixes

The syntax specification in the Definition is somewhat sloppy, leaving a number of ambiguities and minor issues. We provide the details to resolve the relevant ones. Mostly, these just blesses existing practice in SML implementations. See Appendix A for a motivation and detailed discussion of the issues.

Changes to the Definition

Section 2.2 (Special constants):

- In the paragraph defining formatting characters, add carriage return and vertical tab to the list of non-printable characters included.

Section 2.6 (Infixed operators):

- In the 1st paragraph, extend the sentence starting with "The only required use of `op`..." by inserting the following before the semicolon:

  > [...] in an expression or pattern;

Section 3.4 (Grammar for Modules):

- In Figure 6, add the following note:

  > Restriction: A declaration dec appearing in a structure declaration may not be a sequential or local declaration.

- In Figure 7, add the following note:

  > Restriction: In a sequential specification, $spec_2$ may not contain a sharing specification.

- In Figure 8, extend the restriction with the following sentence:

  > Furthermore, the $strdec$ may not be a sequential declaration $strdec_1$ ⟨`;`⟩ $strdec_2$.

Section 8 (Programs):

- Extend the comment on rule 187:

  [...], except for possible fixity directives contained in the $topdec$.

Appendix A (Derived Forms):

- In Figure 17, add respective indices $1..m$ to the ty annotations appearing on both sides of the definition of the function value binding form.

Appendix B (Full Grammar):

- Extend the first sentence as follows:

  [...], together with the derived form of Figure 18 in Appendix A.

- Add the following to the third paragraph:

  The same applies to patterns, where the extra classes AppPat and InfPat are introduced, yielding

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

- In the third bullet, replace the paragraph starting with "Note particularly that..." with:

  Note that the use of precedence does not prevent a phrase, which is an instance of a form with higher precedence, having a constitutent which is an instance of a form with lower precendence, as long as they can be resolved unambiguously. Thus for example

  ```
  if ... then while ... do ... else while ... do ...
  ```

  is quite admissible and parses as

  ```
  if ... then (while ... do ...) else (while ... do ...)
  ```

  However, precedence rules out phrases which cannot be disambiguated without violating precedence, such as

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

  This change should allow the use of simple precedence rules as provided by Yacc to disambiguate parsing.

- In Figure 21, replace the production for $fvalbind$ with the following productions:

  $$\begin{array}{lcll} fvalbind & ::= & fmatch\ \langle \texttt{and}\ fvalbind \rangle & \\ fmatch & ::= & fmrule\ \langle \texttt{|}\ fmatch \rangle & \\ fmrule & ::= & fpat\ \langle \texttt{:}\ ty \rangle\ \texttt{=}\ exp & \\ fpat & ::= & \langle \texttt{op} \rangle vid\ atpat_1\ \cdots\ atpat_n & n \geq 1 \\ & & \texttt{(}\ atpat_1\ vid\ atpat_2\ \texttt{)}\ atpat_3\ \cdots\ atpat_n & n \geq 3 \\ & & atpat_1\ vid\ atpat_2 & \end{array}$$

  Furthermore, add the following note:

  Restriction: The expressions $exp_1, \ldots, exp_{m-1}$ in a $fvalbind$ may not terminate in a match.

- In Figure 22, replace the productions for pat with the following:

apppat	::=	*atpat*	
		⟨`op`⟩*longvid atpat*	constructed value
infpat	::=	*apppat*	
		$infpat_1$ *vid* $infpat_2$	constructed value (infix)
pat	::=	*infpat*	
		pat `:` *ty*	typed
		⟨`op`*vid* ⟨`:` *ty*⟩ `as` *pat*	layered

Compatibility

These are merely fixes, they do not change the language beyond resolving ambiguities. The only exception is the restriction on nesting matches in a *fvalbind*, which is what all SML systems implement anyway.

B.2 Semantic Fixes

The Definition contains a number of bugs in inference rules and other parts of the formal semantics. Some of them undermine soundness, some are just plain typos. The changes we propose merely plug these holes and bless existing practice, they should not have any further effect on the defined language. See Appendix A for motivation and detailed discussion of the issues.

Note: Along with the changes described in the following sections, the only known (non-pedantic) issue remaining is the lack of a requirement for type sharing to be consistent with respect to the involved constructor environments, which makes exhaustiveness and irredundancy of patterns an ill-defined concept. No straightforward fix seems to exist within the Definition's formal framework, short of introducing a global consistency requirement similar to SML'90.

Changes to the Definition

Section 2.9 (Syntactic Restrictions):

- Add the following bullet:

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

Section 4.10 (Inference Rules):

- Add a closing parenthesis to the conclusion of Rule 28.

Section 4.11 (Further Restrictions):

- In the first bullet, in the first sentence change "the program context" to

  > [...] the program context consisting of the smallest enclosing declaration

Section 5.7 (Inference Rules):

- Add the following side condition to rule 64:

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

- Add the following side condition to rule 78:

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

Section 7.2 (Compound Objects):

- In the definition of the operator $\downarrow$: Env$\times$Int $\to$ Env, replace the triple "$(SI,\ TE,\ VI)$" with "$(SI,\ TI,\ VI)$".

Section 7.3 (Inference Rules):

- In rule 182, replace both occurences of IB with B.
- In the conclusion of rules 184–186, replace $B'\langle' \rangle$ with $B'\langle +B \rangle$.

Appendix D (The Initial Dynamic Basis):

- Add the following paragraph:

  Furthermore, the initial state is defined by

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

Appendix E (Overloading):

- In the last paragraph of the introduction, change the last sentence to:

  For this purpose, the surrounding text is the smallest enclosing declaration.

Appendix E.1 (Overloaded special constants):

- Before the sentence starting with "Special constants...", insert the following sentence:

  The class Real may not contain type names that admit equality.

Appendix E.2 (Overloaded value identifiers):

- In Figure 27, change the types of <, >, <=, >= to:

```
numtxt * numtxt -> bool
```

Compatibility

These are merely fixes, they do not change the language beyond plugging holes.

B.3 Monomorphic Non-exhaustive Bindings

In order maintain to soundness of polymorphic typing in the presence of effects, polymorphism is restricted to non-expansive bindings. Non-expansiveness is a syntactic condition on expressions that is sufficient to guarantee absence of effects (including exceptions) during their evaluation.

However, an exception may still occur if the pattern in the binding is not exhaustive. That behaviour is somewhat inconsistent, and more importantly, unnecessarily complicates typed compilation schemes, like used by several SML compilers (see Appendix A.3).

Non-exhaustive patterns are ruled out in polymorphic bindings. That is, pathological programs like

```
val x::xs = []
```

but also

```
val x::xs = [NONE, NONE]
```

are no longer valid. Such declarations are rather useless, and can easily be rewritten.

Changes to the Definition

Section 4.8 (Non-expansive Expressions):

- Change the rules for obtaining $\alpha^{(k)}$ to:

$$\alpha^{(k)} = \begin{cases} \text{tyvars}\,\tau \setminus \text{tyvars}\,C, & \text{if } pat \text{ exhaustive and } exp \text{ non-expansive in } C; \\ (), & \text{otherwise.} \end{cases}$$

- Add the following sentence:

  A pattern is *exhaustive* if it matches all values (of the right type, cf. Section 4.11).

Compatibility

This is not a conservative change, but very unlikely to break any practical program. It is already implemented in SML/NJ and TILT.

B.4 Simplified Recursive Value Bindings

The current syntax for recursive value declarations allows many phrases that are either useless or confusing. For example,

```
val rec rec rec f = fn x => x
val f = fn x => x and rec g = fn x => f x
```

Note that in the latter declaration, the right-hand side of `g` does not refer to the `f` of the same declaration.

The syntax can be simplified by only allowing `rec` directly after the `val` keyword.

Furthermore, the Definition currently allows recursive value declarations to overwrite identifier status. This is inconsistent with the rules of the dynamic semantics, and hence arguably a bug (see Appendix A.5). It also is counter-intuitive and a nuisance to implement (no implementation does it "correctly"). This possibility is removed. The change is simplified by reversing the order of the `rec` keyword and an eventual type variable sequence in a value declaration.

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 4, replace the production for value declarations with:

$$[dec ::=] \qquad \texttt{val}\ \langle\texttt{rec}\rangle\ tyvarseq\ valbind \qquad \text{value declaration}$$

- Remove the second production for $valbind$.

Section 2.9 (Syntactic Restrictions):

- In the 4th bullet, replace the start of the sentence with:

  For each value binding pat `=` exp in a value declaration with `rec`, [...]

Section 4.10 (Inference Rules):

- Change rule 15 to:

$$\frac{\begin{array}{l}U = \text{tyvars}(tyvarseq) \qquad \langle\text{tynames } VE \subseteq T \text{ of } C\rangle \\ \langle\forall vid \in \text{Dom } VE,\ vid \notin \text{Dom } C \text{ or } is \text{ of } C(vid) = \texttt{v}\rangle \\ C + U\langle + VE\rangle \vdash valbind \Rightarrow VE \qquad VE' = \text{Clos}_{C,valbind}\, VE \qquad U \cap \text{tyvars } VE' = \emptyset\end{array}}{C \vdash \texttt{val}\ \langle\texttt{rec}\rangle\ tyvarseq\ valbind \Rightarrow VE' \text{ in Env}} \tag{15}$$

- Remove rule 26. Add the respective comment to the comment on rule 15, but replace the last two sentences with the following:

  > The side condition on the value identifiers in C ensures that $C + VE$ does not overwrite identifier status in the recursive case. For example, the program "`datatype t = f; val rec f = fn x => x;`" is not legal.

Section 6.6 (Function Closure):

- In the second paragraph, replace "recursive value bindings of the form `rec` $valbind$" with "recursive value declarations of the form `val rec` $valbind$".

Section 6.7 (Inference Rules):

- Change rule 114 to:

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \texttt{val}\ \langle\texttt{rec}\rangle\ tyvarseq\ valbind \Rightarrow \langle\text{Rec}\rangle\, VE \text{ in Env}} \tag{114}$$

- Remove rule 126.

Appendix A (Derived Forms):

- In the third paragraph, replace "`val` *tyvarseq* `rec` *valbind*" with "`val rec` *tyvarseq valbind*".
- In Figure 17, the box for declarations, replace the transformed form of function declarations with:

`fun` *tyvarseq fvalbind*	`val rec` *tyvarseq fvalbind*

Appendix B (Full Grammar):

- In Figure 21, replace the production for value declarations with:

$$[dec ::=] \qquad \texttt{val}\ \langle\texttt{rec}\rangle\ tyvarseq\ valbind \qquad \text{value declaration}$$

- Remove the second production for *valbind*.

Compatibility

The change intentionally rules out some previously legal programs and reverses the order in which the `rec` keyword and the optional type variable sequence may appear in a value declaration. However, at least one major SML implementation - namely SML/NJ - always implemented the revised syntax, so the change is unlikely to affect existing programs.

No current implementation follows the Definition with respect to overwriting of identifier status (although they deviate in different ways). Consequently, this part of the change is even less likely to affect existing programs.

B.5 Abstype as Derived Form

Abstype is a leftover from SML's pre-module days and is now fully subsumed by structures and sealing. Besides being redundant, the current specification of abstype is incoherent with respect to equality (see Appendix A.3), an issue for which no obvious fix exists.

Although abstype is practically unused in modern code, it cannot be removed without breaking backwards compatibility. Turning it into a derived form avoids this problem, while still simplifying the bare language and resolving the coherence issues.

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 4, remove the production for `abstype`.

Section 4.9 (Type Structures and Type Environments):

- Remove the last paragraph.

Section 4.10 (Inference Rules):

- Remove Rule 19 and the corresponding comments.

Section 6.7 (Inference Rules):

- Remove Rule 118.

Appendix A (Dervied Forms):

- In Figure 17, add the following rewriting rule before the existing one for `abstype`:

`abstype` *datbind* `with` *dec* `end`	`local datatype` *datbind* `in`
	  `type` *typbind*′ `;` *dec*
	`end`

  and extend the note to

  (see note in text concerning *datbind*′ and *typbind*′)

- In the bullet list in the text referring to Figure 17, add the following item:

  In the abstype form, *typbind*′ is obtained from *datbind* by replacing all right-hand sides by the corresponding left-hand side, i.e. "*tyvarseq tycon* = *conbind* ⟨| *datbind*⟩" becomes "*tyvarseq tycon* = *tyvarseq tycon* ⟨| *typbind*′⟩"

Compatibility

This is a conservative change. The new specification is slightly more permissive than the original static semantics of `abstype`, because the equality attribute of the defined type is no longer hidden. However, this is precisely what is necessary to fix the aforementioned coherence issues.

While the change may marginally affect the abstraction properties of code still using abstype, it can be argued that the obsolete nature of `abstype` makes this neglectable in practice.

The change simplifies implementations, because it enables them to isolate their treatment of `abstype` in the parser.

B.6 Fixed Manifest Type Specifications

For technical reasons, manifest type specifications are defined as a derived form. However, the definition of this form results in scoping rules that are at odds with the rest of the language (see Appendix A.8). The definition of the derived form is changed to eliminate the singularity.

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 19, replace the first two rules with the following one:

type $tyvarseq_1$ $tycon_1$ = ty_1 and ... ... and $tyvarseq_n$ $tycon_n$ = ty_n	include sig type $tyvarseq_1$ $tycon_1$ and ... ... and $tyvarseq_n$ $tycon_n$ end where type $tyvarseq_1$ $tycon_1$ = ty_1 where type ... ... where type $tyvarseq_n$ $tycon_n$ = ty_n

Compatibility

This change breaks programs relying on the current scoping rules. However, since these rules are rather counter-intuitive, not implemented by all SML implementations (Moscow ML and Poly/ML deviate), and they make using `and` in type specifications pointless anyway, we expect those programs to be rare. It is trivial to adapt them to the change.

Only few SML implementations actually implement manifest type specifications as a derived form. The change hence should be a simplification for the majority of implementations, as it removes an annoying singularity in the language rules.

B.7 Abolish Sequenced Type Realisations

The SML syntax allows several type constraints on a signature to be connected with `and`, as in

```
S where type t1 = ty1
    and type t2 = ty2
```

This syntax is hard to parse and only few implementations bother to do it correctly, it is at odds with the rest of the language, and it is useless, because writing another `where` instead of `and` has the very same effect (see Appendix A.8). The syntax does not seem to be widely used either, it is hence abolished.

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 19, remove the box for signature expressions.

Compatibility

The change breaks all programs using the derived form. Adapting affected programs is trivial.

B.8 Line Comments

Under most circumstances, line comments are more convenient to write and to layout than block comments. SML lacks line comments.

The comment marker `(*)` introduces a comment that stretches to the end of the line:

```
fun f x = bla    (*) my function
fun g x = blo    (*) my second function
```

Line comments properly nest into conventional block comments, so the following is one single comment, even though the inner line comment contains a closing comment bracket:

```
(*
fun f x = bla    (*) my function *)
*)
```

Changes to the Definition

Section 2.3 (Comments):

- Reformulate whole section as follows:

  > A *comment* is either *line comment* or a *block comment*. A line comment is any character sequence between the comment delimiter `(*)` and the following end of line. A block comment is any character sequence within comment brackets `(* *)` in which other comments are properly nested. No space is allowed between the characters that make up a comment bracket `(*)` or `(*` or `*)`. An unmatched `(*` should be detected by the compiler.

Compatibility

This extension breaks SML programs containing block comments that have a closing parenthesis `)` as the first character after the opening bracket. Such comments are expected to be extremely rare in existing code, and can easily be modified.

B.9 Extended Literal Syntax

SML currently provides no way to group digits in numeric literals, which makes long numbers hard to read. Underscores are allowed within literals to group digits and increase readability. For example,

```
val pi = 3.141_592_653_596
val billion = 1_000_000_000
val nibbles = 0wx_f300_4588
```

Moreover, SML lacks a notation for binary literals and hence requires fallback to hexadecimal. A C-style notation with a ”0b” prefix enables writing binary literals:

```
val ten = 0b1010
val bits = 0wb1101_0010_1111_0010
```

Note that binary literals particularly benefit from the ability to group digits.

Last, in SML it is a pointless hurdle to remember the order of the different parts in literal prefixes. The order of the different parts in literal prefixes is made arbitrary, allowing `0xw` and `0bw` as synonyms for `0wx` and `0wb`.

Changes to the Definition

Section 2.2 (Special constants):

- Extend the first sentence as follows:

  > [...] and the underscore (_) that neither starts nor ends with an underscore.

- Extend the second sentence:

  > [...] and the underscore that does not end with an underscore.

- Add the following sentence to the end of the paragraph:

  > An integer constant (in binary notation) is an optional negation symbol followed by a non-empty sequence of binary digits `0`, `1` and the underscore that does not end with an underscore.

- Extend the first sentence of the second paragraph as follows:

  > [...] and the underscore not ending with an underscore.

- In the second sentence, replace "is `0wx`" with "is `0wx` or `0xw`".

- Extend the second sentence as follows:

  > [...] and the underscore not ending with an underscore.

- After the second sentence, add:

  > A word constant (in binary notation) is `0wb` or `0bw` followed by a non-empty sequence of binary digits `0`,`1` and the underscore not ending with an underscore.

- Modify the next sentence by replacing "and one or more decimal digits" with:

  > and a sequence of one or more decimal digits and underscores that contains at least one digit

- Add to the the list of examples in the next sentence:

  > `3.141_592_653 3._678_098__E20`

- Add to the list of non-examples:

  > `1_.5 1._E2`

Compatibility

This extension is not conservative, as it may change the meaning of programs that contain literals and wildcards without separating spaces, as in

```
fun f 3_4 = 0
```

or, likewise, programs that put a literal next to an identifier `xw`, `b`, `wb`, or `bw`. However, such programs are highly unlikely to exist in practice.

The scanning functions from the Basis library should be extended to reflect the change by supporting underscores in their input.

B.10 Record Punning

SML allows record patterns of the form {a=a, b=b} to be abbreviated conveniently as {a, b} – sometimes called "punning". The same abbreviation is not currently provided for record expressions. Such an abbreviation can be equally convenient, e.g. for constructing records from local variables:

```
fun circle(x,y,r) =
    let
        val x = ref x and y = ref y and r = ref r
        fun pos() = (!x,!y)
        fun radius() = !r
        fun move(dx,dy) = (x := !x+dx, y := !y+dy)
        fun scale s = (r := !r*s)
    in
        {pos, radius, move, scale}
    end
```

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 15, add the following box:

Expression Rows *exprow*

$vid\ \langle$`:` $ty\rangle\ \langle$`,` $exprow\rangle$	vid = $vid\ \langle$`:` $ty\rangle\ \langle$`,` $exprow\rangle$

Appendix B (Full Grammar):

- In Figure 20, add the following production:

$[exprow ::=]$ $\quad vid\ \langle$`:` $ty\rangle\ \langle$`,` $exprow\rangle$ $\quad$ label as variable

Compatibility

This is a conservative extension.

B.11 Record Extension

When using records, it is sometimes necessary to construct new records from existing ones, by adding only a small number of fields. Similarly, it can be convenient to be able to construct a new record by removing a small number of fields. Currently, SML provides no convenient way of expressing this.

Row capture Row capture is supported by raising the status of the ellipsis `...` in record patterns to make it analogous to a normal field name. The ellipsis refers to all the other fields that have not been named explicitly.

Example:

```
val {d=x, p=y, ...=r} = e
```

This value binding takes the result of expression e, which must be some record that has at least fields d and p, and takes it apart. As usual, it binds the values of the d and p fields to x and y, respectively. But in addition it also binds r to a freshly constructed record value that consists of all the fields of e except d and p.

Example:

```
val {d=x, p=y, ...=r} =
    {a=1, c=3.0, d=nil, f=[1], p="hello", z=NONE}
```

binds x to nil, y to "hello", and r to {a=1, c=3.0, f=[1], z=NONE}.

Record extension Functional record extension is supported by allowing ellipses in record expressions. This restores a sense of "perfect symmetry" between record patterns and record expressions.

Example:

```
{d=e1, p=e2, ...=e3}
```

Here e3 is required to be of record type without fields d and p. The result of the above expression is a record which consists of all the fields that were present in the result of e3 as well as a field d whose type and value are determined by e1 and a field p whose type and value are determined by e2.

Example:

```
let val r = {a=1, c=3.0, f=[1], z=NONE}
in {d=nil, p="hello", ...=r}
end
```

This expression yields

```
{a=1, c=3.0, d=nil, f=[1], p="hello", z=NONE}
```

Record type extension Like record values, record types can be constructed by extension.

Example:

```
type 'a t = {a : 'a, b : bool}
type 'a u = {c : char, d : 'a list, ... : 'a t}
```

Again, ellipses denote the type that is to be extended. It must be a record type. The result is a record type which consists of the combined fields. The example yields

```
type 'a u = {a : 'a, b : bool, c : char, d : 'a list}
```

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 3, change the production for pattern row wildcards to:

$$[patrow ::=] \qquad \texttt{...} = pat \qquad \text{ellipses}$$

  Add the following production for type-expression rows:

$$[tyrow ::=] \qquad \texttt{...} : ty \qquad \text{ellipses}$$

- In Figure 4, add the following production for expression rows:

$$[exprow ::=] \qquad \texttt{...} = exp \qquad \text{ellipses}$$

Section 2.9 (Syntactic Restrictions):

- Remove the first bullet ruling out repeated labels.

Section 4.2 (Compound Objects):

- Add the following definition after the paragraph defining modification of maps:

  The restriction of a map f by a set S, written $f \setminus S$, is defined as

$$f \setminus S = \{x \mapsto f(x); x \in \text{Dom } f \setminus S\}$$

Section 4.7 (Non-expansive Expressions):

- Add the following production for non-expansive expression rows:

$$[nexprow ::=] \qquad \texttt{...} = nexp$$

Section 4.10 (Inference Rules):

- Change Rule 6 to:

$$\frac{C \vdash exp \Rightarrow \tau \qquad \langle C \vdash exprow \Rightarrow \varrho \qquad lab \notin \text{Dom } \varrho \rangle}{C \vdash lab \texttt{ = } exp \ \langle \texttt{,} \ exprow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \varrho \rangle} \tag{6}$$

- Add the following rule:

$$\frac{C \vdash exp \Rightarrow \varrho \text{ in Type}}{C \vdash \texttt{...} = exp \Rightarrow \varrho} \tag{6a}$$

- Change Rules 38 and 39 as follows:

$$\frac{C \vdash pat \Rightarrow (VE, \varrho \text{ in Type})}{C \vdash \texttt{...} = pat \Rightarrow (VE, \varrho)} \tag{38}$$

$$\frac{\begin{array}{c} C \vdash pat \Rightarrow (VE, \tau) \\ \langle C \vdash patrow \Rightarrow (VE', \varrho) \qquad \text{Dom } VE \cap \text{Dom } VE' = \emptyset \qquad lab \notin \text{Dom } \varrho \rangle \end{array}}{C \vdash lab \texttt{ = } pat \ \langle \texttt{,} \ patrow \rangle \Rightarrow (VE \langle + VE' \rangle, \{lab \mapsto \tau\} \langle + \varrho \rangle)} \tag{39}$$

  Remove the comment regarding Rule 39.

- Change Rule 49 to:

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash tyrow \Rightarrow \varrho \qquad lab \notin \text{Dom } \varrho \rangle}{C \vdash lab \ \texttt{:} \ ty \ \langle \texttt{,} \ tyrow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \varrho \rangle} \tag{49}$$

  Remove the respective comment.

- Add the following rule:

$$\frac{C \vdash ty \Rightarrow \varrho \text{ in Type}}{C \vdash \texttt{...} \ \texttt{:} \ ty \Rightarrow \varrho} \tag{49a}$$

Section 4.11 (Further Restrictions):

- Change the first item to:

  > For each occurence of a record expression containing ellipses, i.e. of the form $\{lab_1\texttt{=}exp_1\texttt{,} \ldots \texttt{,} lab_m\texttt{=}exp_m\texttt{,} \ \texttt{...=}exp_0\}$ the program context must determine uniquely the domain $\{lab_1, \ldots, lab_n\}$ of its row type, where $m \leq n$; thus, the context must determine the labels $\{lab_{m+1}, \ldots, lab_n\}$ of the fields of exp_0. Likewise for record patterns containing ellipses. For these purposes, explicit type constraints may be needed.

Section 6.7 (Inference Rules):

- Add the following rule:

$$\frac{E \vdash exp \Rightarrow r \text{ in Val}}{E \vdash \texttt{...} \ \texttt{=} \ exp \Rightarrow r} \tag{95a}$$

- Change Rule 140 to:

$$\frac{E, r \text{ in Val} \vdash pat \Rightarrow VE/\text{FAIL}}{E, r \vdash \texttt{...} \ \texttt{=} \ pat \Rightarrow VE/\text{FAIL}} \tag{140}$$

- Change Rule 142 to:

$$\frac{E, r(lab) \vdash pat \Rightarrow VE \qquad \langle E, r \setminus \{lab\} \vdash patrow \Rightarrow VE'/\text{FAIL} \rangle}{E, r \vdash lab \ \texttt{=} \ pat \ \langle \texttt{,} \ patrow \rangle \Rightarrow VE \langle + VE'/\text{FAIL} \rangle} \tag{142}$$

Appendix A (Derived Forms):

- In Figure 15, add a box for expression rows:

  Expression Rows $exprow$

$\texttt{...} \ \texttt{=} \ exp\texttt{,} \ exprow$	$\texttt{let val} \ vid \ \texttt{=} \ exp \ \texttt{in} \ \{exprow\texttt{,} \ \texttt{...} \ \texttt{=} \ vid\} \ \texttt{end}$

  (see note in text concerning $exprow$; vid new)

- In Figure 16, extend the box for pattern rows as follows:

...	... = _
... $\langle$= $pat\rangle$, $patrow$	$patrow$, ... $\langle$= $pat\rangle$

  (see note in text concerning $patrow$)

  Add a box for type-expression rows:

  Type-expression Rows $tyrow$

$\texttt{...} \ \texttt{:} \ ty\texttt{,} \ tyrow$	$tyrow\texttt{,} \ \texttt{...} \ \texttt{:} \ ty$

  (see note in text concerning $tyrow$)

- Add the following paragraph:

  > Note that the derived forms for ellipses in the middle of expression rows, pattern rows or type-expression rows are only valid if they can be transformed to bare syntax. This implies that the remaining rows may not again contain ellipses.

Appendix B (Full Grammar):

- In Figure 20, add the following production for expression rows:

  $[exprow ::=]$ `...` `=` $exp\ \langle$`,` $exprow\rangle$ ellipses

- In Figure 22, change the production for pattern row wildcards to:

  $[patrow ::=]$ `...` $\langle$`=` $pat\rangle\ \langle$`,` $patrow\rangle$ ellipses

- In Figure 23, add the following production for type-expression rows:

  $[tyrow ::=]$ `...` `:` $ty\ \langle$`,` $tyrow\rangle$ ellipses

Compatibility

This is a conservative extension. Type inference is not entirely straightforward in the given form, but the issues are only slightly harder than those already caused by the existing ellipsis mechanism (unresolved row variables become shared between different record types and hence require additional propagation). Type inference actually becomes simpler in the presence of SML#-style record polymorphism, but an efficient implementation of the dynamic semantics becomes somewhat trickier.

B.12 Record Update

When using records, it is often necessary to construct new records from existing ones, by changing only a small number of fields. For example, this happens when using records to express functional objects, or in the use of records to encode default arguments. Currently, SML provides no convenient way to express this.

Record update is supported with a new derived form $\{atexp$ `where` $exprow\}$. The keyword `where` is chosen such that it plays a similar role as it does in the signature language. The syntax is designed such that it adheres to the principle of least surprise, is economic, and convenient.

Changes to the Definition

Appendix A (Derived Forms):

- Extend the box for expressions as follows:

$\{atexp$ `where` $\langle exprow\rangle\}$	`let val` $\{\langle patrow$`,`$\rangle$ `...` `=` $vid\}$ `=` $atexp$ `in` $\{\langle exprow$`,`$\rangle$ `...` `=` $vid\}$ `end`

  (see note in text concerning $patrow$; vid new)

- Add the following paragraph after the second of the section:

In the derived forms for record update, $patrow$ is obtained from $exprow$ by replacing all right-hand sides by wildcards. Note that $exprow$ may not contain ellipses.

Appendix B (Full Grammar):

- In Figure 20, change the production for record expressions to:

$$[exp ::=] \qquad \{\ \langle atexp\ \texttt{where}\rangle\ \langle exprow\rangle\ \} \qquad \text{record}$$

Compatibility

This is a conservative extension. Its specification relies on record extension, as defined in the previous section.

B.13 Conjunctive Patterns

SML provides layered patterns vid `as` pat to allow naming a value and simultaneously matching its structure. The name must be put first. However, depending on the situation, it often is more convenient to put the name last.

Instead of adding a second syntactic form, we propose generalizing layered patterns to arbitrary conjunctive patterns pat_1 `as` pat_2, which trivially supports both forms, while also eliminating grammar problems that exist with the current syntax (it is not LR(1)).

Conjunctive patterns are particularly useful in combination with nested matches (see Appendix B.15).

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 3, replace the production for layered patterns with:

$$[pat ::=] \qquad pat_1\ \texttt{as}\ pat_2 \qquad \text{conjunctive}$$

Section 4.10 (Inference Rules):

- Replace rule 43 with:

$$\frac{C \vdash pat_1 \Rightarrow (VE_1, \tau) \qquad C \vdash pat_2 \Rightarrow (VE_2, \tau) \qquad \text{Dom } VE_1 \cap \text{Dom } VE_2 = \emptyset}{C \vdash pat_1\ \texttt{as}\ pat_2 \Rightarrow (VE_1 + VE_2, \tau)} \tag{43}$$

Section 4.11 (Further Restrictions):

- Add the following bullet:

  Every pattern of the form pat_1 `as` pat_2 must be consistent, i.e., there must exist at least one value that is matched by both patterns.

Section 6.7 (Inference Rules):

- Replace rule 149 with:

$$\frac{E, v \vdash pat_1 \Rightarrow VE_1 \qquad E, v \vdash pat_2 \Rightarrow VE_2/\text{FAIL}}{E, v \vdash pat_1 \texttt{ as } pat_2 \Rightarrow (VE_1 + VE_2)/\text{FAIL}} \tag{149}$$

- Add the following rule:

$$\frac{E, v \vdash pat_1 \Rightarrow \text{FAIL}}{E, v \vdash pat_1 \texttt{ as } pat_2 \Rightarrow \text{FAIL}} \tag{149a}$$

Appendix A (Derived Forms):

- In Figure 16, remove the box for pattern rows.

Appendix B (Full Grammar):

- In Figure 22, replace the production for layered patterns with:

$$[pat ::=] \qquad pat_1 \texttt{ as } pat_2 \qquad \text{conjunctive}$$

Compatibility

This is a conservative extension. Pattern matching is not complicated significantly by the change. It actually simplifies parsing.

B.14 Disjunctive Patterns

Disjunctive patterns pat_1 `|` pat_2 avoid the need for repeating the same right-hand side in a match several times, by allowing to fold multiple left-hand side patterns into one. In certain cases this can significantly reduce code size, as well as the temptation to write fragile catch-all clauses to get around the code duplication.

Note that the syntax immediately supports writing multiple alternatives pat_1 `|` $\ldots$ `|` pat_n, as well as "multiple" matches:

```
case exp of
    A | B | C => 1
  | D | E => 2
```

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 3, add the following production:

$$[pat ::=] \qquad pat_1 \texttt{ | } pat_2 \qquad \text{disjunctive}$$

Section 2.9 (Syntactic Restrictions):

- Add the following comment to the 2nd bullet:

> [...] (identifiers appearing in both branches of a disjunctive pattern are bound only once)

Section 4.10 (Inference Rules):

- Add the following rule for patterns:

$$\frac{C \vdash pat_1 \Rightarrow (VE, \tau) \qquad C \vdash pat_2 \Rightarrow (VE, \tau)}{C \vdash pat_1 \texttt{ | } pat_2 \Rightarrow (VE, \tau)} \tag{43a}$$

Section 4.11 (Further Restrictions):

- In item 2, insert the following sentence after the first one:

> Similarly, in a disjunctive pattern of the form pat_1 `|` pat_2, the second pattern must match some value not matched by the first one. Moreover, either of them must match some value that is not matched by the surrounding pattern or match rule.

The wording regarding irredundancy does require compilers to warn about cases like `fn _|3 => ()`, but not `fn 3|_ => ()`, although the latter is redundant as well. None of the compilers currently supporting disjunctive patterns seems to detect the latter, and it is not obvious how to extend the usual algorithm appropriately.

Section 6.7 (Inference Rules):

- Add the following rules for patterns:

$$\frac{E, v \vdash pat_1 \Rightarrow VE}{E, v \vdash pat_1 \texttt{ | } pat_2 \Rightarrow VE} \tag{149b}$$

$$\frac{E, v \vdash pat_1 \Rightarrow \text{FAIL} \qquad E, v \vdash pat_2 \Rightarrow VE/\text{FAIL}}{E, v \vdash pat_1 \texttt{ | } pat_2 \Rightarrow VE/\text{FAIL}} \tag{149c}$$

Appendix B (Full Grammar):

- In Figure 21, add the following production (as the last one, giving least precedence):

$$[pat ::=] \qquad pat_1 \texttt{ | } pat_2 \qquad \text{disjunctive}$$

Compatibility

This is a conservative extension.

B.15 Nested Matches

Patterns may contain nested matching constructs of the form

$$pat_1 \texttt{ with } pat_2 \texttt{ = } exp$$

Such a *nested match* is matched by first matching pat_1, then evaluating exp, and matching its result against pat_2. Variables bound in pat_1 may occur in exp. The pattern fails when either pattern does not match. The pattern binds the combined set of variables occuring in pat_1 and pat_2. For instance, consider:

```
case xs of [x,y] with SOME z = f(x,y) => x+y+z | _ => 0
```

If `xs` is a two-element list `[x,y]` such that `f(x,y)` returns `SOME z`, then the whole expression evaluates to `x+y+z`, otherwise to `0`.

Nested matches are a very general construct. They can be useful in combination with disjunctive patterns,

```
case args of x::_ | (nil with x = 0) => ...
```

or with guards (see Appendix B.16):

```
fun escape #"\"" = "\\\""
  | escape #"\\" = "\\\\"
  | escape (c with n=ord c) if (n < 32) = "\\^" ^ str(chr(n+64))
  | escape c = str c
```

The main importance of nested matches, however, is that they form the basis to uniformly define pattern guards (Appendix ext-guards) as well as a simple form of "views" (Appendix B.17) as syntactic sugar.

In patterns with multiple subpatterns, nested matches to the right may refer to variables bound by patterns to the left. See Appendix B.17 for examples.

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 3, add the following production for patterns:

  $[pat ::=]$ $\quad pat_1$ `with` pat_2 `=` exp $\quad$ nested match

  and the note

  > *Restriction:* The pattern pat_1 in a nested match pat_1 `with` pat_2 `=` exp may not itself be a nested match, unless enclosed by parentheses.

- In Figure 4, add the following note:

  > *Restriction:* The pattern pat in a $valbind$ may not be of the form pat_1 `with` pat_2 `=` exp, unless enclosed by parentheses.

Section 4.7 (Non-expansive Patterns):

- Add the following paragraph:

  > A pattern is *non-expansive* if it does not contain a nested match of the form pat_1 `with` pat_2 `=` exp.

Section 4.8 (Closure):

- Add the following additional side condition to the first case defining $\text{Clos}_{C,valbind}\, VE(vid)$:

  > if pat is non-expansive, ...

Section 4.10 (Inference Rules):

- In rule 39, change the premise "$C \vdash patrow \Rightarrow (VE', \varrho)$" to "$C + VE \vdash patrow \Rightarrow (VE', \varrho)$".
- Likewise, in rule 43 (as given in Appendix B.13), change the premise "$C \vdash pat_2 \Rightarrow (VE_2, \varrho)$" to "$C + VE_1 \vdash pat_2 \Rightarrow (VE_2, \varrho)$".
- Add the following rule for patterns:

$$\frac{\begin{array}{cc} C \vdash pat_1 \Rightarrow (VE_1, \tau) & C + VE_1 \vdash pat_2 \Rightarrow (VE_2, \tau') \\ C + VE_1 \vdash exp \Rightarrow \tau' & \text{Dom } VE_1 \cap \text{Dom } VE_2 = \emptyset \end{array}}{C \vdash pat_1 \texttt{ with } pat_2 \texttt{ = } exp \Rightarrow (VE_1 + VE_2, \tau)} \quad (43b)$$

Section 4.11 (Further Restrictions):

- Add the following sentence to the 2nd bullet:

  > For the purpose of checking exhaustiveness, any contained nested match, pat_1 `with` pat_2 `=` exp may be assumed to fail, regardless of the form of exp, except if pat_2 is exhaustive itself. Further note that exp may contain side effects and hence change the content of references that have already been matched.

Section 6.7 (Inference Rules):

- In rule 142, change the premise "$E, r \vdash patrow \Rightarrow VE'/\text{FAIL}$" to "$E + VE, r \vdash patrow \Rightarrow VE'/\text{FAIL}$".
- Likewise, in rule 43 (as given in Appendix B.13), change the premise "$E, v \vdash pat_2 \Rightarrow VE_2/\text{FAIL}$" to "$E + VE_1, v \vdash pat_2 \Rightarrow VE_2/\text{FAIL}$".
- Add the following rules for patterns:

$$\frac{E, v \vdash pat_1 \Rightarrow \text{FAIL}}{E, v \vdash pat_1 \texttt{ with } pat_2 \texttt{ = } exp \Rightarrow \text{FAIL}} \quad (149d)$$

$$\frac{E, v \vdash pat_1 \Rightarrow VE_1 \qquad E + VE_1 \vdash exp \Rightarrow v' \qquad E + VE_1, v' \vdash pat_2 \Rightarrow VE_2/\text{FAIL}}{E, v \vdash pat_1 \texttt{ with } pat_2 \texttt{ = } exp \Rightarrow VE_1 + VE_2/\text{FAIL}} \quad (149e)$$

Appendix B (Full Grammar):

- In Figure 21, add the following note:

  > *Restriction:* The pattern pat in a $valbind$ may not be of the form pat_1 `with` pat_2 `=` exp, unless enclosed by parentheses.

- Add the following production for patterns:

  $$[pat ::=] \qquad pat_1 \texttt{ with } pat_2 \texttt{ = } exp \qquad \text{nested match}$$

  and the note

  > *Restriction:* The pattern pat in a $valbind$ may not be of the form pat_1 `with` pat_2 `=` exp, unless enclosed by parentheses.

Compatibility

Except for the new reserved word `?`, this is a mostly conservative extension. Due to potential side effects in guard conditions, it renders pattern matching impure. This has a particular consequence on patterns of the form `ref` $atpat$, whose behaviour may depend on the evaluation of previous nested matches. In particular, the following case expression,

```
case (i, r) of
    (_, ref true) => 1
  | (2, _) with _ = f() => 2
  | (_, ref false) => 3
```

is not an exhaustive match, since `r` may be `false`, but could get set to `true` during evaluation of `f()`.

Note that conjunctive patterns "pat_1 `as` pat_2" could also be defined as a derived form for

$$vid \texttt{ with } pat_1 \texttt{ = } vid \texttt{ with } pat_2 \texttt{ = } vid$$

but that would alter the meaning of exhaustiveness.

B.16 Pattern Guards

Pattern guards avoid code duplication by letting pattern matching fall through if a particular condition is not met. This is not possible by merely using conditionals on the right-hand side.

Pattern guards are introduced as a simple derived form for nested matches:

pat `if` exp

They are also allowed with function-value bindings:

```
fun min x y if (x < y) = x
  | min x y            = y
```

Note that in this case the guard condition needs to be an atomic expression, in order to avoid syntactic ambiguity.

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 16, add the following boxes for patterns pat:

pat `if` exp	pat `with true =` exp

- In Figure 17, extend the box for Function-value Bindings by adding

  $\langle$`if` $atexp_i\rangle$

  (with $i = 1..m$) to each equation in the left box, as the last component of the left-hand sides, and likewise to each match in the right box, as the last component before `=>`.

Appendix B (Full Grammar):

- In Figure 21, extend the production for *fmrule* (as defined in Appendix B.1) as follows:

$$fmrule \quad ::= \quad fpat\ \langle \texttt{:}\ ty \rangle\ \langle \texttt{if}\ atexp \rangle\ \texttt{=}\ exp$$

- Extend the restriction note added by the change from Appendix B.15 by inserting the following before "unless enclosed by parentheses":

  [...] or pat `if` exp [...]

- In Figure 22, add the following production for patterns:

$$[pat ::=] \qquad pat\ \texttt{if}\ exp \qquad \text{guard}$$

  and extend the restriction note added by the change from Appendix B.15 by inserting the following before "unless enclosed by parentheses":

  [...] or pat `if` exp [...]

Compatibility

This is a conservative extension over nested matches. It is mostly conservative over plain SML (see Appendix B.15).

B.17 Transformation Patterns

The main importance of nested matches, is that they form the basis to uniformly define a simple form of "poor man's views" as syntactic sugar, which we refer to as *transformation patterns*:

`?`exp
`?`exp pat

The first form provides boolean "views":

```
fun skipSpace(?isSpace :: cs) = skipSpace cs
  | skipSpace cs = cs
```

The parameterised form allows actual matching. Consider an ADT for queues:

```
type 'a queue
val empty : 'a queue
val enqueue : 'a * 'a queue -> 'a queue
val dequeue : 'a queue -> ('a * 'a queue) option
```

With such patterns, queues can be pattern matched as follows:

```
fun process (?dequeue(x,q)) = (digest x; process q)
  | process _ = terminate()
```

A transformation may be denoted by an arbitrary expression, giving rise to *dynamic transformations*. Consider a simple set ADT:

```
type set
val empty : set
val insert : int -> set -> set
val isempty : set -> bool
val has : int -> set -> bool
```

The following is possible:

```
fun f n ?isempty = f1 ()
  | f n ?(has n) = f2 ()
  | f n _ = f3 ()
```

Or another example, with a parameterised dynamic transformation:

```
(*) val split : char -> string -> (string * string) option

fun manExp(?(split #"E")(m,e)) = (m,e)
  | manExp s = (s,"1")
```

As a minor subtelety, in patterns with multiple subpatterns, nested matches and transformation patterns to the right may refer to variables bound by patterns to the left. For example,

```
(x, ?(equals x))
x as ?(notOccurs x)(T(x1,x2))
```

In particular, this allows the function `f` above to be expressed more without a separate `case` expression.

Note that, in addition to transformation patterns, HaMLet-S also features proper views (Appendix B.20). While it is probably undesirable to have both features in a finalised language, simultaneous support in an experimental system like ours allows evaluating the merits of each approach.

Changes to the Definition

Section 2.1 (Reserved Words):

- Add `?` to the list of reserved words.

Section 2.9 (Syntactic Restrictions):

- Add `NONE` and `SOME` to the list of value identifiers that may not be re-bound.

Appendix A (Derived Forms):

- In Figure 16, add the following boxes for patterns pat:

`?`$atexp$	vid `with true =` $atexp$ vid	(vid new)
`?`$atexp$ $atpat$	vid `with SOME` $atpat$ `=` $atexp$ vid	(vid new)

Appendix B (Full Grammar):

- In Figure 22, add the following production for atomic patterns:

$$[atpat ::=] \qquad \texttt{?}\ atexp \qquad \text{transformation}$$

- Add the following production for application patterns (as introduced by the changes described in Appendix B.1):

$$[apppat ::=] \qquad \texttt{?}\ atexp\ atpat \qquad \text{constructed transformation}$$

Appendix C (The Initial Static Basis):

- Add `option` to the definition of T_0.
- In Figure 24, add the following entry:

$$\begin{array}{rcll} \texttt{option} & \mapsto & (\ \texttt{option}, & \{\texttt{NONE} \mapsto (\forall\, '\texttt{a}\,.\,'\texttt{a option}, \texttt{c}), \\ & & & \ \ \texttt{SOME} \mapsto (\forall\, '\texttt{a}\,.\,'\texttt{a} \to '\texttt{a option}, \texttt{c})\}) \end{array}$$

- In Figure 25, add the following entries to the left column:

$$\begin{array}{rcl} \texttt{NONE} & \mapsto & (\forall\, '\texttt{a}\,.\,'\texttt{a option}, \texttt{c}) \\ \texttt{SOME} & \mapsto & (\forall\, '\texttt{a}\,.\,'\texttt{a} \to '\texttt{a option}, \texttt{c}) \end{array}$$

Appendix D (The Initial Dynamic Basis):

- Add "$\texttt{NONE} \mapsto (\texttt{NONE}, \texttt{c})$" and "$\texttt{SOME} \mapsto (\texttt{SOME}, \texttt{c})$" to the definition of VE_0.
- In Figure 26, add the following entry:

$$\texttt{option} \ \mapsto \ \{\texttt{NONE} \mapsto (\texttt{NONE}, \texttt{c}),\ \texttt{SOME} \mapsto (\texttt{SOME}, \texttt{c})\}$$

Compatibility

Except for the new reserved word `?`, this is a mostly conservative extension (see Appendix B.15).

B.18 Optional Bars and Semicolons

SML syntax separates match clauses with a bar `|`. The usual coding convention is to lay out matches such that the bar comes before each clause. However, the first clause is an unpleasant special case:

```
case exp0 of
    pat1 => exp1
  | pat2 => exp2
  | pat3 => exp3
```

Taking aesthethic considerations aside, the assymmetry between the cases is a nuisance for editing, because clauses cannot be reordered by a simple cut & paste operation.

An additional bar is allowed to optionally appear before the first clause, such that the above can be written as:

```
case exp0 of
  | pat1 => exp1
  | pat2 => exp2
  | pat3 => exp3
```

For consistency, the same extension is made for function value bindings, and for datatype declarations. For instance,

```
datatype 'a exp =
  | Const  of 'a
  | Var    of string
  | Lambda of string * 'a exp
  | App    of 'a exp * 'a exp
```

In a similar vein, optional terminating semicolons are allowed for expression sequences. For example, in a let expression:

```
fun myfunc2(x, y) =
    let
      val z = x + y
    in
      f x;
      g y;
      h z;
    end
```

The same applies to parenthesised expressions and sequences.

Changes to the Definition

Section 2.8 (Grammar):

- In Figure 4, change the productions for exception handling and functions to, respectively:

$[exp ::=]$	exp `handle` $\langle$`|`$\rangle$ $match$	handle exception
	`fn` $\langle$`|`$\rangle$ $match$	function

- Change the production for datatype bindings to:

$$datbind \quad ::= \quad tyvarseq\ tycon\ \texttt{=}\ \langle\texttt{|}\rangle\ conbind\ \langle\texttt{and}\ datbind\rangle$$

Section 3.4 (Grammar for Modules):

- In Figure 7, change the productions for datatype descriptions to:

$$datdesc \quad ::= \quad tyvarseq\ tycon\ \texttt{=}\ \langle\texttt{|}\rangle\ condesc\ \langle\texttt{and}\ datdesc\rangle$$

Section 4.10 (Inference Rules):

- Adapt the syntax in the conclusion of rules 10, 12 and 28 appropriately.

Section 5.7 (Inference Rules):

- Adapt the syntax in the conclusion of rule 81 appropriately.

Section 6.7 (Inference Rules):

- Adapt the syntax in the conclusion of rules 104–106, 108 and 128 appropriately.

Section 7.3 (Inference Rules):

- Adapt the syntax in the conclusion of rule 178 appropriately.

Appendix A (Derived Forms):

- In Figure 15, change the rule for case expressions to

`case` exp `of` $\langle$`|`$\rangle$ $match$	`(fn` $\langle$`|`$\rangle$ $match$ `)(`exp`)`

- Change the left-hand side of the rule for sequential expressions to:

`(`exp_1 `;` $\cdots$ `;` exp_n `;` exp $\langle$`;`$\rangle$`)`

- Add a box as follows:

`(`exp `;` `)`	`(`exp `)`

- Change the left-hand side of the rule for let expressions to:

`let` dec `in` exp_1 `;` $\cdots$ `;` exp_n $\langle$`;`$\rangle$ `end`

- In Figure 17, change the first line in the definition of function clauses to:

$\langle$`|`$\rangle$ $\langle$`op`$\rangle vid\ atpat_{11} \cdots atpat_{1n}$ $\langle$`:` $ty_1\rangle$ `=` exp_1 $[\ldots]$	$[\ldots]$ $\langle$`|`$\rangle$ `(` $atpat_{11}$`,` $\ldots$`,` $atpat_{1n}$ `)` `=>` exp_1 $\langle$`:` $ty_1\rangle$ $[\ldots]$

- In Figure 21, change the production for datatype bindings to:

$$datbind \quad ::= \quad tyvarseq\ tycon\ \texttt{=}\ \langle\texttt{|}\rangle\ conbind\ \langle\texttt{and}\ datbind\rangle$$

Appendix B (Full Grammar):

- In Figure 20, change the productions for sequences and let expressions to:

$[atexp ::=]$	`(`exp_1 `;` $\cdots$ `;` exp_n $\langle$`;`$\rangle$`)`	sequence, $n \geq 1$
	`let` dec `in` exp_1 `;` $\cdots$ `;` exp_n $\langle$`;`$\rangle$ `end`	local declaration, $n \geq 1$

- Remove the production for parenthesised expressions.
- Change the productions for exception handling, functions, and case expressions to, respectively:

$[exp ::=]$	exp `handle` $\langle$`|`$\rangle$ $match$	handle exception
	`fn` $\langle$`|`$\rangle$ $match$	function
	`case` exp `of` $\langle$`|`$\rangle$ $match$	case analysis

- In Figure 21, change the production for datatype bindings to:

$$datbind \quad ::= \quad tyvarseq\ tycon\ \texttt{=}\ \langle\texttt{|}\rangle\ conbind\ \langle\texttt{and}\ datbind\rangle$$

- In Figure 21, change the production for $fvalbind$ (as defined in Appendix B.1) to:

$$[fvalbind ::=] \quad \langle\texttt{|}\rangle\ fmatch\ \langle\texttt{and}\ fvalbind\rangle$$

Compatibility

This is a conservative extension.

B.19 Optional `else` Branch

With imperative code it is often convenient to be allowed to omit the `else` branch of a conditional:

`if` exp_1 `then` exp_2

This is a simple derived form. The type of exp_2 has to be `unit` if the `else` branch is omitted. As usual, dangling `else` phrases associate to the innermost `if`.

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 15, add a second rule for conditionals:

`if` exp_1 `then` exp_2	`if` exp_1 `then` exp_2 `else` $()$

Appendix B (Full Grammar):

- Append the following bullet:

  Likewise, a conditional `if` exp_1 `then` ... extends as far right as possible; thus, optional `else` branches group with the innermost conditional.

- In Figure 20, change the productions for conditionals to:

$[exp ::=]$ `if` exp_1 `then` exp_2 $\langle$`else` $exp_3\rangle$ conditional

Compatibility

This is a conservative extension.

B.20 Views

One of the most wanted features for SML (and other functional languages) are *views*. Views enable the definition of abstract constructors for arbitrary types that can be used in patterns as if they were ordinary datatype constructors.

A view primarily defines a set of constructors and two functions for converting between these and the actual type the view is defined for. For example, consider a simple view allowing (positive) integers to be viewed as inductive numbers:

```
viewtype peano = int as Zero | Succ of int
with
   fun from Zero     = 0
```

```
      | from (Succ n) = n+1
   fun to 0         = Zero
     | to n if (n>0) = Succ(n-1)
     | to n         = raise Domain
end
```

This defines a view for type `int`. The type constructor `peano` provides a name for this view. Views may be defined for arbitrary types, and there may be arbitrarily many views for a given type.

Given the viewtype definition above, we can construct integers using the constructors it introduces:

```
val n = Succ(Succ(Succ Zero))   (*) binds n to 3
val n = Succ 2                  (*) likewise
```

The function `from` given with the view declaration defines how a view constructor is converted to the underlying type, and is applied implicitly for every occurrence of a view constructor in an expression.

The inverse function `to` defines how a value of the underlying type is interpreted in terms of the view constructors. It is applied implicitly whenever a value of the underlying type is matched against a pattern using one of the view's constructors:

```
fun fac Zero    = 1
  | fac(Succ n) = Succ n * fac n
```

This defines a factorial function on integers. When `fac` is applied to an integer i, the function `to` is implicitly applied to i first and its result is matched against the constructors appearing in the definition of `fac`.

The body of a view declaration may contain arbitrary (auxiliary) declarations, but must feature the two functions `from` and `to` with the appropriate types. None of the declarations is visible outside the view declaration.

Views must be used *consistently*, that is, a match may not use different views, or a view and concrete constants of the underlying type, *for the same position* in a pattern. For instance, the following is illegal:

```
fun fac (0 | 1)  = 1
  | fac (Succ n) = Succ n * fac n
```

Thanks to this restriction, the compiler is still able to check exhaustiveness and irredundancy of patterns, even in the presence of views.

Views are particularly interesting in conjunction with abstract types. For that purpose, it is possible to specify views in signatures:

```
signature COMPLEX =
sig
   type complex
   viewtype cart = complex as Cart of real * real
   viewtype pole = complex as Pole of real * real
end
```

A view specification can either be matched by a corresponding view declaration, or by an appropriate datatype definition:

```
structure Complex :> COMPLEX =
struct
   datatype cart = Cart of real * real
   type complex = cart
   viewtype pole = complex as Pole of real * real
   with
      open Math
      fun to(Cart(x,y)) = Pole(sqrt(x*x + y*y), atan2(x,y))
      fun from(Pole(r,t)) = Cart(r*cos(t), r*sin(t))
   end
end
```

The implementation of a viewtype is kept abstract, and both of the above views can be used uniformly where appropriate:

```
open Complex
fun add(Cart(x1,y1), Cart(x2,y2)) = Cart(x1+x2, y1+y2)
fun mul(Pole(r1,t1), Pole(r2,t2)) = Pole(r1*r2, t1+t2)
```

Instead of opening the structure, a view can also be pulled into scope (and thus enable unqualified use of its constructors) by a viewtype replication declaration, analogous to SML's datatype replication:

```
viewtype cart = viewtype Complex.cart
```

Apart from viewtype replication, the name of a view acts as a synonym for the underlying representation type – except inside the view definition itself, where it used to denote the (otherwise anonymous) datatype representing the view.

More extensive examples can be found in `doc/examples/views.sml`.

The design of views was inspired mainly by the papers of Wadler [W87] and Okasaki [O98]. The main differences are the following:

- Views are named, to enable proper interplay with the module system, particularly view replication.
- Unlike Okasaki's proposal, views are bidirectional, that is, they can be used to symmetrically construct *and* deconstruct values.
- Unlike both proposals, views may not be mixed, thus still enabling standard pattern checks.
- Unlike both proposals, view definitions are not recursive. In particular, the view constructors cannot be used as a view within its own definition.
- Unlike Okasaki's proposal, the formal definition below does not support memoization. This could probably be added by means of informal comments.

Changes to the Definition

Section 2.1 (Reserved Words):

- Add `viewtype` to the list of reserved words.

Section 2.8 (Grammar):

- In Figure 4, add the following production for declarations:

$$[dec ::=] \quad \texttt{viewtype}\ tyvarseq\ tycon\ \texttt{=}\ ty\ \texttt{as}\ \langle\texttt{|}\rangle\ conbind\ \texttt{with}\ dec\ \texttt{end} \quad \text{viewtype}$$

Section 2.9 (Syntactic Restrictions):

- Extend the second bullet with:

  [...] or the $conbind$ of a `viewtype` declaration.

- Extend the bullet added by the changes described in Appendix B.2 as follows:

  [...]; similarly, in a declaration of the form "`viewtype` $tyvarseq\ tycon$ `=` ty `as` $conbind$ `with` dec `end`", any $tyvar$ occuring in ty or $conbind$ must occur in $tyvarseq$.

Section 3.4 (Grammar for Modules):

- In Figure 7, add the following production for specifications:

$$[spec ::=] \quad \texttt{viewtype}\ tyvarseq\ tycon\ \texttt{=}\ ty\ \texttt{as}\ \langle\texttt{|}\rangle\ condesc \quad \text{viewtype}$$

Section 3.5 (Syntactic Restrictions):

- Extend the second bullet with:

  [...] or the $condesc$ of a `viewtype` spcification.

- Replace the latter half of the fourth bullet with:

  [...]; similarly, in a specification of the form "`viewtype` $tyvarseq\ tycon$ `=` ty `as` $condesc$" or a signature expression of the form "$sigexp$ `where type` $tyvarseq\ longtycon$ `=` ty", any $tyvar$ occuring in ty or $condesc$ must occur in $tyvarseq$.

Section 4.2 (Compound Objects):

- In Figure 10, Change the definition of value environments to:

$$\begin{aligned} VE &\in \text{ValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{TypeScheme} \times \text{ValStatus} \\ vs &\in \text{ValStatus} = \text{IdStatus} \cup \text{TyName} \end{aligned}$$

- In the last paragraph of the text, replace "an identifier status" with "a value status" and all occurrences of is with vs.

- In the last sentence, replace "or an *exception constructor*" with "an *exception constructor* or a *view constructor*" and replace "`v`, `c` or `e`" with "`v`, `c`, `e` or a type name t".

Section 4.7 (Non-expansive Expressions):

- In the *Restriction*, replace "is of $C(longvid) \in \{\texttt{c}, \texttt{e}\}$" with "$vs$ of $C(longvid) \neq$ `v`".

Section 4.8 (Closure):

- Replace all occurrences of is with vs.

Section 4.9 (Type Structures and Type Environments):

- Extend the first sentence with:

  [...], or there is a type name t such that for all $(\sigma, vs) \in \text{Ran } VE$, $vs = t$.

Section 4.10 (Inference Rules):

- In rule 2, 15 (as modified by change described in Appendix B.4), 34 and 35, replace occurrences of is with vs.
- Add the following rule for declarations:

$$\frac{\begin{array}{c} t \notin T \text{ of } C \qquad \text{arity } t = k \qquad t \text{ does not admit equality} \\ tyvarseq = \alpha^{(k)} \qquad C \vdash ty \Rightarrow \tau \qquad C, \alpha^{(k)} t \vdash conbind \Rightarrow VE \\ C \oplus (\text{Clos } VE, \{tycon \mapsto (t, \text{Clos } VE)\}) \vdash dec \Rightarrow E \\ \sigma \text{ of } E(\texttt{from}) \succ \forall \alpha^{(k)}.\alpha^{(k)} t \to \tau \qquad \sigma \text{ of } E(\texttt{to}) \succ \forall \alpha^{(k)}.\tau \to \alpha^{(k)} t \\ VE' = \{vid \mapsto (\sigma\{\Lambda\alpha^{(k)}.\tau/t\}, t) \;;\; (\text{Clos } VE)(vid) = (\sigma, \texttt{c})\} \\ TE = \{tycon \mapsto (\Lambda\alpha^{(k)}.\tau,\ VE')\} \end{array}}{C \vdash \texttt{viewtype}\ tyvarseq\ tycon\ \texttt{=}\ ty\ \texttt{as}\ conbind\ \texttt{with}\ dec\ \texttt{end} \Rightarrow (VE',\ TE) \text{ in Env}} \quad (17a)$$

  and add a comment:

  (17a) Unlike a datatype, a viewtype is not recursive. *Comment:*

Section 4.11 (Further Restrictions):

- Add a fourth point:

  4. The compiler must issue an error if a match or a pattern in a value binding makes inconsistent use of view constructors, such that there might exist a value that, in a single matching operation, has to be matched against view constructors of different view types, or against a view constructor and a pattern that is not a view. For example, if `C` and `D` are view constructors of different views for type `int`, then the patterns "`C | D`" or "`2 as C`" are invalid, likewise the match "`(_,C) =>` ... `| (_,D) =>` ...". This restriction ensures that the checks described in the previous points are always possible.

Section 5.5 (Enrichment):

- In the third point defining $E_1 \succ E_2$, replace all occurrences of is with vs and replace the line defining enrichment on identifier status with:

  $vs_1 = vs_2$ or $vs_2 = \mathtt{v}$ or $vs_1 = \mathtt{c}$ and $vs_2 = t$

- Replace the second point defining $(\theta_1, VE_1) \succ (\theta_2, VE_2)$ with:

  2. Either $VE_2 = \{\}$, or $VE_1 = VE_2$, or $VE_1 = \{vid \mapsto (\sigma, \mathtt{c})\ ;\ VE_2(vid) = (\sigma, vs)\}$

Section 5.7 (Inference Rules):

- In rule 2, 34 and 35, replace all occurrences of is with vs.
- Add the following rule for specifications:

$$\frac{\begin{array}{c} t \notin T \text{ of } B \qquad \text{arity } t = k \qquad t \text{ does not admit equality} \\ tyvarseq = \alpha^{(k)} \qquad C \text{ of } B \vdash ty \Rightarrow \tau \qquad C \text{ of } B, \alpha^{(k)} t \vdash condesc \Rightarrow VE \\ VE' = \{vid \mapsto (\sigma\{\Lambda\alpha^{(k)}.\tau/t\}, t)\ ;\ (\text{Clos}\, VE)(vid) = (\sigma, \mathtt{c})\} \\ TE = \{tycon \mapsto (\Lambda\alpha^{(k)}.\tau,\ VE')\} \end{array}}{B \vdash \mathtt{viewtype}\ tyvarseq\ tycon\ \mathtt{=}\ ty\ \mathtt{as}\ condesc \Rightarrow (VE', TE) \text{ in Env}} \quad (71a)$$

Section 6.1 (Reduced Syntax):

- In the first bullet, remove "constructor and".
- Replace the second bullet with:

  All equations "$\mathtt{=}\ ty$" are omitted from `viewtype` declarations.

Section 6.3 (Compound Objects):

- In Figure 13, change the definition of value environments to:

$$\begin{array}{rcl} VE & \in & \text{ValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{Val} \times \text{ValStatus} \\ vs & \in & \text{ValStatus} = \text{IdStatus} \cup (\text{Val} \times \text{VId}) \end{array}$$

Section 6.7 (Inference Rules):

- In rule 91, replace is with vs.
- Add the following rule for declarations:

$$\frac{\begin{array}{c} \vdash conbind \Rightarrow VE \qquad E + (VE, \{tycon \mapsto VE\}) \vdash dec \Rightarrow E' \\ v \text{ of } E'(\mathtt{from}), v \text{ of } E'(\mathtt{to}) \vdash conbind \Rightarrow VE' \qquad TE = \{tycon \mapsto VE'\} \end{array}}{E \vdash \mathtt{viewtype}\ tyvarseq\ tycon\ \mathtt{as}\ conbind\ \mathtt{with}\ dec\ \mathtt{end} \Rightarrow (VE', TE) \text{ in Env}} \quad (116a)$$

- In rule 129, add "$\langle \mathtt{of}\ ty \rangle$" to the phrase in the conclusion, and replace existing single brackets "$\langle \ldots \rangle$" with double brackets "$\langle\langle \ldots \rangle\rangle$".

- Add the following rules for constructor bindings:

$$\frac{\{\texttt{from} \mapsto (v_{\texttt{from}}, \texttt{v}), vid' \mapsto (vid, \texttt{v})\} \text{ in Env} \vdash \texttt{from}\ vid' \Rightarrow v \qquad vid' \neq \texttt{from} \qquad \langle v_{\texttt{from}}, v_{\texttt{to}} \vdash conbind \Rightarrow VE\rangle}{v_{\texttt{from}}, v_{\texttt{to}} \vdash vid\ \langle \texttt{|}\ conbind\rangle \Rightarrow \{vid \mapsto (v, (v_{\texttt{to}}, vid))\}\langle + VE\rangle \text{ in Env}} \tag{129a}$$

$$\frac{v = (vid''\ \texttt{=>}\ \texttt{from}\ \texttt{(}\ vid'\ vid''\ \texttt{)}, E, \{\}) \text{ in Val} \quad \texttt{from} \neq vid' \neq vid'' \neq \texttt{from} \qquad E = \{\texttt{from} \mapsto (v_{\texttt{from}}, \texttt{v}), vid' \mapsto (vid, \texttt{v})\} \text{ in Env} \quad \langle v_{\texttt{from}}, v_{\texttt{to}} \vdash conbind \Rightarrow VE\rangle}{v_{\texttt{from}}, v_{\texttt{to}} \vdash vid\ \texttt{of}\ ty\ \langle \texttt{|}\ conbind\rangle \Rightarrow \{vid \mapsto (v, (v_{\texttt{to}}, vid))\}\langle + VE\rangle \text{ in Env}} \tag{129b}$$

and add a comment:

(129a),(129b) In these and the rules 137a, 137b and 147a, 147b the choice of vid' and vid'' is arbitrary, up to the side conditions stated in the rules. *Comment:*

- In rule 135, replace is with vs.
- Add the following rules for atomic patterns:

$$\frac{E(longvid) = (v', (v_{\texttt{to}}, vid)) \qquad vid' \neq \texttt{to} \qquad \{\texttt{to} \mapsto (v_{\texttt{to}}, \texttt{v}), vid' \mapsto (v, \texttt{v})\} \text{ in Env} \vdash \texttt{to}\ vid' \Rightarrow vid}{E, v \vdash longvid \Rightarrow \{\}} \tag{137a}$$

$$\frac{E(longvid) = (v', (v_{\texttt{to}}, vid)) \qquad vid' \neq \texttt{to} \qquad \{\texttt{to} \mapsto (v_{\texttt{to}}, \texttt{v}), vid' \mapsto (v, \texttt{v})\} \text{ in Env} \vdash \texttt{to}\ vid' \Rightarrow v'' \qquad v'' \neq vid}{E, v \vdash longvid \Rightarrow \text{FAIL}} \tag{137b}$$

- Add the following rules for patterns:

$$\frac{E(longvid) = (v', (v_{\texttt{to}}, vid)) \qquad vid' \neq \texttt{to} \qquad \{\texttt{to} \mapsto (v_{\texttt{to}}, \texttt{v}), vid' \mapsto (v, \texttt{v})\} \text{ in Env} \vdash \texttt{to}\ vid' \Rightarrow (vid, v'') \qquad E, v'' \vdash atpat \Rightarrow VE/\text{FAIL}}{E, v \vdash longvid\ atpat \Rightarrow VE/\text{FAIL}} \tag{147a}$$

$$\frac{E(longvid) = (v', (v_{\texttt{to}}, vid)) \qquad vid' \neq \texttt{to} \qquad \{\texttt{to} \mapsto (v_{\texttt{to}}, \texttt{v}), vid' \mapsto (v, \texttt{v})\} \text{ in Env} \vdash \texttt{to}\ vid' \Rightarrow v'' \qquad v'' \notin \{vid\} \times \text{Val}}{E, v \vdash longvid\ atpat \Rightarrow \text{FAIL}} \tag{147b}$$

Section 7.1 (Reduced Syntax):

- In the first bullet, remove "constructor and".

Section 7.2 (Compound Objects):

- In Figure 14, change the definition of value interfaces to:

$$\begin{array}{lcl} VI & \in & \text{ValInt} = \text{VId} \xrightarrow{\text{fin}} \text{ValIntStatus} \\ vis & \in & \text{ValIntStatus} = \text{IdStatus} \cup \{\texttt{f}\} \end{array}$$

- Change the definition of Inter : ValEnv $\to$ ValInt to:

$$\begin{aligned}\text{Inter}(VE) \quad &= \quad \{vid \mapsto is \;;\; VE(vid) = (v, is)\} \\ &\quad + \{vid \mapsto \texttt{f} \;;\; VE(vid) = (v, (v', vid''))\}\end{aligned}$$

  and extend the following sentence with:

  > [...] and abstracting view constructors with `f`.

- Change the definition of $\downarrow$: ValEnv $\times$ ValInt $\to$ ValEnv to:

$$\begin{aligned}VE \downarrow \text{ValInt} \quad &= \quad \{vid \mapsto (v, is) \;;\; VE(vid) = (v, vs) \text{ and } VE(vid) = is\} \\ &\quad + \{vid \mapsto (v, vs) \;;\; VE(vid) = (v, vs) \text{ and } VE(vid) = \texttt{f}\}\end{aligned}$$

- In the parenthesised sentence following, replace "identifier status" with "value status" and add:

  > [...], except in the case of view constructors.

Section 7.3 (Inference Rules):

- Add the following rule for specifications:

$$\frac{\vdash condesc \Rightarrow VI \qquad VI' = \{vid \mapsto \texttt{f} \;;\; VI(vid) = \texttt{c}\} \qquad TI = \{tycon \mapsto VI'\}}{IB \vdash \texttt{viewtype}\ tyvarseq\ tycon\ \texttt{as}\ condesc \Rightarrow (VI', TI) \text{ in Int}} \tag{169a}$$

- In rule 179, add "$\langle$`of` $ty\rangle$" to the phrase in the conclusion, and replace existing single brackets "$\langle \ldots \rangle$" with double brackets "$\langle\langle \ldots \rangle\rangle$".

Appendix A (Derived Forms):

- In Figure 17, extend the box for declarations as follows:

`viewtype` $tycon$ `=` `viewtype` $longtycon$	`datatype` $tycon$ `=` `datatype` $longtycon$

- In Figure 19, extend the box for specifications as follows:

`viewtype` $tycon$ `=` `viewtype` $longtycon$	`datatype` $tycon$ `=` `datatype` $longtycon$

Appendix B (Full Grammar):

- In Figure 21, add the following productions for declarations:

$[dec ::=]$	`viewtype` $tyvarseq\ tycon$ `=` ty `as` $\langle$`|`$\rangle\ conbind$	viewtype
	`with` dec `end`	
	`viewtype` $tycon$ `=` `viewtype` $longtycon$	viewtype replication

Compatibility

Apart from the additional keyword `viewtype`, this is a conservative extension.

B.21 Do Declarations

A very frequent idiom in SML are declarations of the form

```
val _ = exp
```

which are used to evaluate an expression for its side effects. This idiom is somewhat verbose and ugly.

On the toplevel, expressions can be evaluated by simply writing them in place of a declaration (which abbreviates a declaration of `it`). However, this form is not available in local scope, and moreover does require putting a semicolon before and after the expression, which is somewhat counterintuitive. This form only is useful in a REPL.

A new derived form simply abbreviates "`val () =`" with the keyword `do`.

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 17, add the following to the Declarations box:

`do` *exp*	`val ( ) =` *exp*

Appendix B (Full Grammar):

- In Figure 21, add the following production for declarations:

$[dec ::=]$ `do` *exp* evaluation

Compatibility

This is a conservative extension.

B.22 Withtype in Signatures

The absence of the `withtype` derived form in signatures clearly is an oversight in the definition. The derived form is as useful in signatures as it is in declarations.

Changes to the Definition

Appendix A (Derived Forms):

- In Figure 19, add the following to the Specifications box:

`datatype` $datdesc$ `withtype` $typbind$	`datatype` $datdesc'$ `;` `type` $typbind$

  and extend the note as follows

  (see the note in text concerning $datdesc'$, $typdesc$, and $longtycon_1, \ldots, longtycon'_m$)

- Append the following paragraph to the text:

In the form involving `withtype`, the identifiers bound by $datdesc$ and by $typbind$ must be distinct. The transformed description $datdesc'$ is obtained from $datdesc$ by expanding out all the definitions made by $typbind$, analogous to $datbind$ above. The phrase "`type` $typbind$" can be reinterpreted as a type specification that is subject to further transformation.

Compatibility

This is a conservative extension, which is already supported by most implementations.

B.23 Higher-order Functors

To support higher-order modules, structure expressions are generalised to include functor expressions, analogous to function expressions in the core:

`fct` $strid$ `:` $sigexp$ `=>` $strexp$

Likewise, signature expressions may denote dependent functor signatures:

`fct` $strid$ `:` $sigexp_1$ `->` $sigexp_2$

As a derived form, non-dependent functor signatures (where $strid$ does not occur in $sigexp_2$) may be abbreviated as follows:

$sigexp_1$ `->` $sigexp_2$

SML's functor declarations are degraded to a mere derived forms, analogous to function declarations with `fun` in the core language. They support curried functors:

`functor` $strid$ `(` $strid_1$ `:` $sigexp_1$ `)` ... `(` $strid_n$ `:` $sigexp_n$ `)` $\langle$`:` $sigexp\rangle$
`=` $strexp$

For uniformity, and to avoid subtle syntax, the identifier classes for structures and functors are merged. As another derived form, SML/NJ compatible syntax is provided for functor descriptions in signatures:

`functor` $strid$ `(` $strid_1$ `:` $sigexp_1$ `)` ... `(` $strid_n$ `:` $sigexp_n$ `)` `:` $sigexp$

Functor application syntax is generalised to

$strexp_1$ $strexp_2$

as in the core. Parentheses are allowed anywhere in structure and signature expressions. The derived form allowing a parenthesised declaration $strdec$ as a functor argument is maintained and generalised by enabling

`(` $strdec$ `)`

to abbreviate a structure in all contexts. For symmetry,

`(` *spec* `)`

can be used to abbreviate a signature. Particularly, it can abbreviate a functor argument:

`fct (`*spec*`) =>` *strexp*
`fct (`*spec*`) ->` *sigexp*

which is also allowed in the functor declaration and specification derived forms, generalising the similar derived form known from SML.

The semantics of higher-order functors is kept simple. All functors are fully generative. The only change to semantic objects of the Definition is in the codomain of stucture environments StrEnv, which may now contain functors.

More extensive examples can be found in `doc/examples/higher-order-functors.sml`.

Changes to the Definition

Section 3.1 (Reserved Words):

- Add `fct` to the list of additional reserved words for modules.

Section 3.2 (Identifiers):

- Replace the first sentence with:

  > The only additional identifier class for Modules is SigId (signature identifiers).

- Replace the start of the second sentence with "Signature identifiers . . . ".

Section 3.4 (Grammar for Modules):

- In Figure 5, remove FunDec and FunBind from the list of phrase classes.
- In Figure 6, replace the *strexp* production "*funid* `(` *strexp* `)`" for functor application with:

  $[strexp ::=]$ $\quad strexp_1\ strexp_2$ functor application (L)

  and add the following productions:

  $[strexp ::=]$ $\quad$ `fct` *strid* `:` *sigexp* `=>` *strexp* $\quad$ functor
  `(` *strexp* `)`

- In Figure 6, add the following *sigexp* productions:

  $[sigexp ::=]$ $\quad$ `fct` *strid* `:` $sigexp_1$ `->` $sigexp_2$ $\quad$ functor
  `(` *sigexp* `)`

- In Figure 8, remove the productions for *fundec* and *funbind*, and the functor declaration production for *topdec*. Change the caption of the figure to "Grammar: Top-level Declarations".

Section 3.5 (Syntactic Restrictions):

- In the first bullet, change "$strbind$, $sigbind$, or $funbind$" to "$strbind$ or $sigbind$".

Section 4.2 (Compound Objects):

- In Figure 10, change the definition of StrEnv as follows:

$$\begin{array}{rcl} SE & \in & \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Mod} \\ M & \in & \text{Mod} = \text{Env} \cup \text{FunSig} \end{array}$$

  Note: a more consistent treatment would include renaming $SE \in$ StrEnv to $ME \in$ ModEnv, but we refrain form that here, in order to keep the number of changes as small as possible.

- To the paragraph referring to Figure 10, add the following sentence:

  > The object class FunSig of functor signatures is defined in Section 5.1.

Section 4.10 (Inference Rules):

- In the last paragraph of the introduction, remove component F from the equation decomposing B, and replace "other components F and G" with "other component G".

Section 5.1 (Semantic Objects):

- In Figure 11, change definitions as follows:

$$\begin{array}{rcl} \Sigma \text{ or } (T)M & \in & \text{Sig} = \text{TyNameSet} \times \text{Mod} \\ \Phi \text{ or } (T)(M, (T')M') & \in & \text{FunSig} = \text{TyNameSet} \times \text{Mod} \times \text{Sig} \end{array}$$

Section 5.3 (Signature Instantiation):

- Replace all occurrences of E with M.

Section 5.4 (Functor Signature Instantiation):

- Replace all pairs of the form "$(E, (T')E')$" with respective forms "$(M, (T')M')$.

Section 5.5 (Enrichment):

- Append the following to item 1:

  > [...], where $M_1 \succ M_2$ either means $M_1 = E_1$ and $M_2 = E_2$ such that E_1 enriches E_2, or $M_1 = \Phi_1$ and $M_2 = \Phi_2$ such that $\Phi_1 \succ \Phi_2$, as defined in Section 5.6.

Section 5.6 (Signature Matching):

- Replace all occurrences of "an environment" with "a module" and E with M.
- Append the following paragraphs:[18]

[18] The defined notion of matching on functor signatures is relatively simplistic. In particular, it make transparent functor signature ascription behave as opaque ascription. For example, the module expression

```
(fct () => (type t = int)) : (fct () -> (type t))
```

will have signature `(fct () -> (type t))`, *not* `(fct () -> (type t = int))` as one might expect. A consistent treatment of transparency is complex in the framework of the Definition and probably not worth the trouble [MT94]. It could be added later as a conservative extension.

A signature $\Sigma_1 = (T_1)M_1$ *matches* a signature $\Sigma_2 = (T_2)M_2$, written $\Sigma_1 \succ \Sigma_2$, if there exists a realisation φ such that $\Sigma_2 \geq \varphi(M_2) \prec M_1$ and $T_1 \cap \text{tynames}\,\Sigma_2 = \emptyset$.

A functor signature $\Phi_1 = (T_1)(M_1, \Sigma_1)$ *matches* a functor signature $\Phi_2 = (T_2)(M_2, \Sigma_2)$, written $\Phi_1 \succ \Phi_2$, if there exists a realisation φ such that $(T_1)M_1 \geq \varphi(M_1) \prec M_2$ and $\varphi(\Sigma_1) \succ \Sigma_2$ and $T_1 \cap \text{tynames}\,\Phi_2 = \emptyset$.

Section 5.7 (Inference Rules):

- In the 2nd paragraph of the introduction, remove component F from the equation decomposing B, and remove "tynames $F\cup$" from the set inequation.
- Change the box giving the form of inference rules for structure expressions to:

$$\boxed{B \vdash \mathit{strexp} \Rightarrow M}$$

  and replace all occurences of E with M in rules 51–53, and E_2 with M in rule 55.
- Change rule 54 to:

$$\frac{\begin{array}{c} B \vdash \mathit{strexp}_1 \Rightarrow \Phi \qquad B \vdash \mathit{strexp}_2 \Rightarrow M \\ \Phi \geq (M'', (T')M'), \;\; M \succ M'' \\ (\text{tynames}\; M \cup T \;\text{of}\; B) \cap T' = \emptyset \end{array}}{B \vdash \mathit{strexp}_1\ \mathit{strexp}_2 \Rightarrow M'} \tag{54}$$

- In the comment for rule 54, replace all occurences of E with M, and replace "$B(\mathit{funid})$" with "Φ".
- Add the following two rules for structure expressions:

$$\frac{\begin{array}{cc} B \vdash \mathit{sigexp} \Rightarrow (T)M & B \oplus \{\mathit{strid} \mapsto M\} \vdash \mathit{strexp} \Rightarrow M' \\ T \cap (T \;\text{of}\; B) = \emptyset & T' = \text{tynames}\; M' \setminus ((T \;\text{of}\; B) \cup T) \end{array}}{B \vdash \texttt{fct}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{=>}\ \mathit{strexp} \Rightarrow (T)(M, (T')M')} \tag{55a}$$

$$\frac{B \vdash \mathit{strexp} \Rightarrow M}{B \vdash \texttt{(}\ \mathit{strexp}\ \texttt{)} \Rightarrow M} \tag{55b}$$

- Change the box giving the form of inference rules for unquantified signature expressions to:

$$\boxed{B \vdash \mathit{sigexp} \Rightarrow M}$$

  and replace all occurences of E with M in rules 61, 63 and 65.
- Add the following two rules for signature expressions:

$$\frac{B \vdash \mathit{sigexp}_1 \Rightarrow (T)M \qquad B \oplus \{\mathit{strid} \mapsto M\} \vdash \mathit{sigexp}_2 \Rightarrow (T')M'}{B \vdash \texttt{fct}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}_1\ \texttt{->}\ \mathit{sigexp}_2 \Rightarrow (T)(M, (T')M')} \tag{64a}$$

$$\frac{B \vdash \mathit{sigexp} \Rightarrow M}{B \vdash \texttt{(}\ \mathit{sigexp}\ \texttt{)} \Rightarrow M} \tag{64b}$$

- In rule 84, replace all occurences of E with M.
- Remove rules 85 and 86.
- Remove rule 89, and change the comment to refer only to rules (87)–(88).

Section 6.3 (Compound Objects):

- In Figure 13, change the definition of structure environments StrEnv as follows:

$$\begin{array}{rcl} SE & \in & \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Mod} \\ M & \in & \text{Mod} = \text{Env} \cup \text{FunctorClosure} \end{array}$$

- Add the following paragraph:

  The object class FunctorClosure represents functors and is defined in Section 7.2.

Section 7.2 (Compound Objects):

- In Figure 14, change the definition of FunctorClosure and Basis as follows:

$$\begin{array}{rcl} (strid : I, strexp : IC, B) & \in & \text{FunctorClosure} = (\text{StrId} \times \text{Int}) \times (\text{StrExp} \times \text{IntConstraint}) \times \text{Basis} \\ IC & \in & \text{IntConstraint} = \text{Int} \cup \{\epsilon\} \\ (G, E) & \in & \text{Basis} = \text{SigEnv} \times \text{Env} \end{array}$$

- Remove the definition for functor environments FunEnv.
- Change SI in the definition of the function Inter as follows:

$$\text{SI} \quad = \quad \{strid \mapsto \text{Inter } M \ ; \ SE(strid) = M\}$$

  and add the following text:

  where Inter M in turn is defined as follows:

$$\text{Inter } M \quad = \quad \begin{cases} \text{Inter } E, & \text{if } M = E; \\ \{\} \text{ in Inter}, & \text{if } M = (strid : I', strexp : IC, B). \end{cases}$$

- Simplify the definition of the function Inter on a basis B to:

$$\text{Inter}(G, E) \quad = \quad (G, \text{Inter } E)$$

- Change the definition of $\downarrow$: StrEnv $\times$ StrInt $\rightarrow$ StrEnv to:

$$SE \downarrow SI \quad = \quad \{strid \mapsto M \downarrow I \ ; \ SE(strid) = M \text{ and } SI(strid) = I\}$$

- After the definition $\downarrow$ on environments, add the following text:

  Here, the definition of $\downarrow$: Mod $\times$ Int $\rightarrow$ Mod is as follows:

$$M \downarrow I \quad = \quad \begin{cases} E \downarrow I, & \text{if } M = E; \\ (strid : I', strexp : I, B), & \text{if } M = (strid : I', strexp : IC, B). \end{cases}$$

  It is extended to interface constraints:

$$M \downarrow IC \quad = \quad \begin{cases} M \downarrow I, & \text{if } IC = I; \\ M, & \text{if } IC = \epsilon. \end{cases}$$

  Interface constraints express optional interface modifications applied to a functor body via higher-order ascription.

Section 7.3 (Inference Rules):

- Change the box giving the form of inference rules for structure expressions to:

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

  and replace all occurences of E and E' with M in rules 151–153 and 155.

- Change rule 154 to:

$$\frac{B \vdash \mathit{strexp}_1 \Rightarrow (\mathit{strid} : I, \mathit{strexp}' : IC, B') \qquad B \vdash \mathit{strexp}_2 \Rightarrow M \qquad B' + \{\mathit{strid} \mapsto M \downarrow I\} \vdash \mathit{strexp}' \Rightarrow M'}{B \vdash \mathit{strexp}_1\ \mathit{strexp}_2 \Rightarrow M' \downarrow IC} \tag{154}$$

- Add the following two rules for structure expressions:

$$\frac{\text{Inter } B \vdash \mathit{sigexp} \Rightarrow I}{B \vdash \texttt{fct}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{=>}\ \mathit{strexp} \Rightarrow (\mathit{strid} : I, \mathit{strexp} : \epsilon, B)} \tag{155a}$$

$$\frac{B \vdash \mathit{strexp} \Rightarrow M}{B \vdash \texttt{(}\ \mathit{strexp}\ \texttt{)} \Rightarrow M} \tag{155b}$$

- Add the following two rules for signature expressions:

$$\frac{IB \vdash \mathit{sigexp}_1 \Rightarrow I_1 \qquad IB + \{\mathit{strid} \mapsto I_1\} \vdash \mathit{sigexp}_2 \Rightarrow I_2}{IB \vdash \texttt{fct}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}_1\ \texttt{->}\ \mathit{sigexp}_2 \Rightarrow I_2} \tag{163a}$$

$$\frac{IB \vdash \mathit{sigexp} \Rightarrow I}{IB \vdash \texttt{(}\ \mathit{sigexp}\ \texttt{)} \Rightarrow I} \tag{163b}$$

- Remove rules 182, 183 and 186.

Appendix A (Derived Forms):

- In Figure 18, replace the box for structure expressions with the following:

Structure Expressions *strexp*

`(` *strdec* `)`	`struct` *strdec* `end`
`fct (` *spec* `) =>` *strexp*	`fct` *strid* `:` `sig` *spec* `end =>` `let open` *strid* `in` *strexp* `end`

(*strid* new)

- In Figure 18, replace the box for functor bindings with the following:

Functor Bindings *funbind*

strid `(` *funarg*$_1$ `)` $\cdots$ `(` *funarg*$_n$ `)` $\langle$`:`$\langle$`>`$\rangle$ *sigexp*$\rangle$ `=` *strexp* $\langle$`and` *funbind*$\rangle$	*strid* `=` `fct` *funarg*$'_1$ `=>` $\cdots$ `fct` *funarg*$'_n$ `=>` *strexp* $\langle$`:`$\langle$`>`$\rangle$ *sigexp*$\rangle$ $\langle$`and` *funbind*$\rangle$

($n \geq 1$; see note in text concerning *funarg*$_1, \ldots,$ *funarg*$'_n$)

- In Figure 18, add a box for structure declarations as follows:

Structure Declarations *strdec*

`functor` *funbind*	`structure` *funbind*

- In Figure 19, add box for functor descriptions as follows:

Functor Descriptions *fundesc*

strid `(` *funarg*$_1$ `)` $\cdots$ `(` *funarg*$_n$ `)` `:` *sigexp*	*strid* `:` `fct` *funarg*$'_1$ `->` $\cdots$ `fct` *funarg*$'_n$ `->` *sigexp*

($n \geq 1$; see note in text concerning $funarg_1, \ldots, funarg'_n$)

- In Figure 19, extend the box for specifications as follows:

`functor` *fundesc*	`structure` *fundesc*

- In Figure 19, extend the box for signature expressions as follows:

`(` *spec* `)`	`sig` *spec* `end`
`fct (` *spec* `) ->` *sigexp*	`fct` *strid* `:` `sig` *spec* `end ->` *sigexp*$'$
sigexp$_1$ `->` *sigexp*$_2$	`fct` *strid* `:` *sigexp*$_1$ `->` *sigexp*$_2$

and add the following note to the box:

(see note in text concerning *sigexp*$'$; *strid* new)

- Add the following paragraph to the text:

In the signature expression form for functors with a specification *spec* as argument, the transformed signature expressions *sigexp*$'$ is obtained from *sigexp* by replacing any identifier *id* that is bound in *spec* with *strid*`.`*id*, except where hidden by a local binding.

- Add the following paragraph to the text:

In the derived forms for functor bindings and functor descriptions, the phrase *funarg* is defined by the following grammar:

$$\begin{array}{lll} funarg & ::= & strid : sigexp \\ & & spec \end{array}$$

In the former case, the corresponding *funarg*$'$ is the same phrase. In the latter case it is the phase "`(` *spec* `)`", such that the meaning is given in terms of the derived form for structure and signature expressions, respectively.

Appendix B (Full Grammar):

- In the 3rd paragraph, extend the first sentence with "and of Modules".
- After the 3rd paragraph, add a paragraph as follows:

There are also three classes of structure expressions as follows:

$$\text{AtStrExp} \subset \text{AppStrExp} \subset \text{StrExp}$$

Finally, there are two classes of signature expressions:

$$\text{AtSigExp} \subset \text{SigExp}$$

- In the next paragraph, replace "Figures 20, 21, 22 and 23" with "Figures 20 to 23d".

- Add the following figure, “Figure 23a: Structure expressions”:

atstrexp	::=	`struct` *strdec* `end`	basic
		`(` *strdec* `)`	basic (short)
		longstrid	structure identifier
		`let` *strdec* `in` *strexp* `end`	local declaration
		`(` *strexp* `)`	
appstrexp	::=	*atstrexp*	
		appstrexp atstrexp	functor application
strexp	::=	*appstrexp*	
		strexp `:` *sigexp*	transparent constraint
		strexp `:>` *sigexp*	opaque constraint
		`fct` *strid* `:` *sigexp* `=>` *strexp*	functor
		`fct (` *spec* `) =>` *strexp*	functor (short)

- Add the following figure, “Figure 23b: Signature expressions”:

atsigexp	::=	`sig` *spec* `end`	basic
		`(` *spec* `)`	basic (short)
		sigid	signature identifier
		`(` *sigexp* `)`	
sigexp	::=	*atsigexp*	
		sigexp `where type`	type realisation
		tyvarseq longtycon `=` *ty*	
		`fct` *strid* `:` *atsigexp* `->` *sigexp*	functor
		`fct (` *spec* `) ->` *sigexp*	functor (short)
		atsigexp `->` *sigexp*	non-dependent functor

- Add the following figure, "Figure 23c: Specifications and descriptions":

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`type` *syndesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc* ⟨`withtype` *typbind*⟩	datatype
		`datatype` *tycon* = `datatype` *longtycon*	replication
		`viewtype` *tyvarseq tycon* = *ty* `as` ⟨`|`⟩ *condesc*	viewtype
		`viewtype` *tycon* = `viewtype` *longtycon*	viewtype replication
		`exception` *exdesc*	exception
		`structure` *strdesc*	structure
		`functor` *fundesc*	functor
		`include` *sigexp*	include
		`include` $sigid_1 \cdots sigid_n$	multiple include
			empty
		spec$_1$ ⟨`;`⟩ *spec*$_2$	sequential
		spec `sharing type`	type sharing
		$longtycon_1$ = $\cdots$ = $longtycon_n$	$(n \geq 2)$
		spec `sharing`	structure sharing
		$longstrid_1$ = $\cdots$ = $longstrid_n$	$(n \geq 2)$

valdesc	::=	*vid* `:` *ty* ⟨`and` *valdesc*⟩
typdesc	::=	*tyvarseq tycon* ⟨`and` *typdesc*⟩
syndesc	::=	*tyvarseq tycon* = *ty* ⟨`and` *syndesc*⟩
datdesc	::=	*tyvarseq tycon* = ⟨`|`⟩ *condesc* ⟨`and` *datdesc*⟩
condesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`|` *condesc*⟩
exdesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`and` *exdesc*⟩
strdesc	::=	*strid* `:` *sigexp* ⟨`and` *strdesc*⟩
fundesc	::=	*strid* `(` $funarg_1$ `)` $\cdots$ `(` $funarg_n$ `)` $(n \geq 1)$ `:` *sigexp* ⟨`and` *fundesc*⟩
funarg	::=	*strid* `:` *sigexp*
		spec

- Add the following figure, "Figure 23d: Structure-level and top-level declarations":

strdec	::=	*dec*	declaration
		`structure` *strbind*	structure
		`functor` *funbind*	functor
		`local` $strdec_1$ `in` $strdec_2$ `end`	local
			empty
		$strdec_1$ ⟨`;`⟩ $strdec_2$	sequential

strbind	::=	*strid* ⟨`:`⟨`>`⟩ *sigexp*⟩ = *strexp* ⟨`and` *strbind*⟩	
funbind	::=	*strid* `(` $funarg_1$ `)` $\cdots$ `(` $funarg_n$ `)` $(n \geq 1)$ ⟨`:`⟨`>`⟩ *sigexp*⟩ = *strexp* ⟨`and` *funbind*⟩	

sigdec	::=	`signature` *sigbind*	
sigbind	::=	*sigid* = *sigexp* ⟨`and` *sigbind*⟩	
topdec	::=	*strdec* ⟨*topdec*⟩	structure-level declaration
		sigdec ⟨*topdec*⟩	signature declaration

Appendix C (The Initial Static Basis):

- In the text, replace "$B_0 = T_0, F_0, G_0, E_0$ where $F_0 = \{\}$, $G_0 = \{\}$ and" with "$B_0 = T_0, G_0, E_0$ where $G_0 = \{\}$ and".

Appendix D (The Initial Dynamic Basis):

- In the text, replace "$B_0 = F_0, G_0, E_0$ where $F_0 = \{\}$, $G_0 = \{\}$ and ..." with "$B_0 = G_0, E_0$ where $G_0 = \{\}$ and ...".

Compatibility

This extension is not conservative because it merges identifier classes for structures and functors. The new reserved word `fct` may also break some existing programs. Otherwise, it is a generalisation of the existing syntax and semantics for modules. Syntactically, it subsumes the higher-order modules of SML/NJ.

B.24 Nested Signatures

In order to make the name spacing mechanism realised by structures applicable to signatures, signatures are allowed as structure members. This implies the presence of qualified signature identifiers, and the addition of signature specifications in signatures:

`signature S =` *sigexp*

A signature definition matches a signature specification if and only if they denote equivalent signatures. Note that – unlike for types – there are no opaque signature specifications, because that would make the type system undecidable in combination with higher-order functors [L97].

Changes to the Definition

The changes described here are relative to the changes for higher-order functors given in Appendix B.23.

Section 3.2 (Identifiers):

- Extend the first sentence with

  [...] and the accompanying *longSigId* (long signature identifiers).

Section 3.4 (Grammar for Modules):

- In Figure 5, remove SigDec form the list of phrase classes, and add the following:

  SigDesc signature descriptions

- In Figure 6, add the following production for structure-level declarations *strdec*:

  [*strdec* ::=] `signature` *sigbind* signature

- Replace the *sigexp* production for signature identifiers to:

  [*sigexp* ::=] *longsigid* signature identifier

- Remove the production for signature declarations $sigdec$.
- In Figure 7, add the following production for specifications:

$$[spec ::=] \qquad \texttt{signature}\ sigdesc \qquad \text{signature}$$

- Add the following production for the new class of signature descriptions:

$$sigdesc \quad ::= \quad sigid\ \texttt{=}\ sigexp\ \langle \texttt{and}\ sigdesc \rangle$$

- In Figure 8, remove the production for signature declarations from $topdec$ and simplify the remaining production for structure declarations to:

$$topdec \quad ::= \quad strdec$$

- Remove the second part of the restriction note that was added by the change from Appendix B.1.

Section 3.5 (Syntactic Restrictions):

- In the second item, replace "or $strdesc$" with ", $strdesc$ or $sigdesc$".

Section 4.2 (Compound Objects):

- In Figure 10, change the definition of environments as follows:

$$E \text{ or } (G, SE, TE, VE) \quad \in \quad \text{Env} = \text{SigEnv} \times \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv}$$

- In the paragraph referring to Figure 10, modify the sentence added by the change described in Appendix B.23 to

> The object classes FunSig of functor signatures and SigEnv of signature environments belong to Modules and are defined in Section 5.1.

Section 4.3 (Projection, Injection and Modification):

- In the paragraph on Modification, replace "$E+(\{\}, \{\}, VE)$" with "$E+(\{\}, \{\}, \{\}, VE)$".

Section 4.10 (Inference Rules):

- In the last paragraph of the introduction, remove the partial sentence after the semicolon, which starts with "one reason [...]".

Section 5.1 (Semantic Objects):

- In Figure 11, simplify the definition of Basis to:

$$B \text{ or } (T, E) \quad \in \quad \text{Basis} = \text{TyNameSet} \times \text{Env}$$

Section 5.5 (Enrichment):

- In the second paragraph, replace "$E_1 = (SE_1, TE_1, VE_1)$" with "$E_1 = (G_1, SE_1, TE_1, VE_1)$"; likewise for E_2.

- Add the following item to the enumeration:

  4. Dom $G_1 \supseteq$ Dom G_2, and $G_1(sigid) \succcurlyeq G_2(sigid)$ for all $sigid \in$ Dom G_2, where $\Sigma_1 \succcurlyeq \Sigma_2$ denotes mutual signature matching as defined in Section 5.6.

Section 5.5 (Signature Matching):

- Extend the second paragraph (as added by the change described in Appendix B.23) with the following sentence:

  We write $\Sigma_1 \succcurlyeq \Sigma_2$ to mean mutual matching $\Sigma_1 \succ \Sigma_2$ and $\Sigma_1 \prec \Sigma_2$.

Section 5.7 (Inference Rules):

- In the 2nd paragraph of the introduction, remove component G from the equation decomposing B, and remove "tynames $G\cup$" from the set inequation.
- Add the following rule for structure declarations:

$$\frac{B \vdash sigbind \Rightarrow G}{B \vdash \texttt{signature}\ sigbind \Rightarrow G \text{ in Env}} \tag{57a}$$

- Change rule 63 as follows:

$$\frac{B(longsigid) = (T)M \qquad T \cap (T \text{ of } B) = \emptyset}{B \vdash longsigid \Rightarrow M} \tag{63}$$

- Remove rule 66.
- Add the following rule for specifications:

$$\frac{B \vdash sigdesc \Rightarrow G}{B \vdash \texttt{signature}\ sigdesc \Rightarrow G \text{ in Env}} \tag{74a}$$

- Add a section for signature description rules of the form

$$\boxed{B \vdash sigdesc \Rightarrow G}$$

  and the following rule:

$$\frac{B \vdash sigexp \Rightarrow \Sigma \qquad \langle B \vdash sigdesc \Rightarrow G \rangle}{B \vdash sigid\ \texttt{=}\ sigexp\ \langle \texttt{and}\ sigdesc \rangle \Rightarrow \{sigid \mapsto \Sigma\}\langle +G \rangle} \tag{84a}$$

- Simplify rule 87 as follows:

$$\frac{B \vdash strdec \Rightarrow E \qquad \text{tyvars } E = \emptyset}{B \vdash strdec \Rightarrow (\text{tynames } E, E) \text{ in Basis}} \tag{87}$$

- Remove rule 88, and change the comment to refer only to rule 87.

Section 6.3 (Compound Objects):

- In Figure 13, change the definition of environments as follows:

$$(G, SE, TE, VE) \text{ or } E \quad \in \quad \text{Env} = \text{SigEnv} \times \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv}$$

- Change the paragraph added by the change described in Appendix B.23 to

  > The object classes FunctorClosure and SigEnv describe functors and signature environments, respectively, and are defined in Section 7.2.

Section 7.2 (Compound Objects):

- In Figure 14, change the definition of interfaces and basis as follows:

$$\begin{array}{rcl} I \text{ or } (G, SI, TI, VI) & \in & \text{Int} = \text{SigEnv} \times \text{StrInt} \times \text{TyInt} \times \text{ValInt} \\ B \text{ or } E & \in & \text{Basis} = \text{Env} \end{array}$$

- Remove the definition of IntBasis.

- In the text, add after the first sentence:

  > A basis B is isomophic to an environment E, but we write explicit injections "E in Basis" and projections "E of B".

  Note: The main motivation here is to keep the number of changes small, as there are many references to the notion of "dynamic basis".

- Adapt the definition of the function Inter : Env $\rightarrow$ Int as follows:

$$\text{Inter}(G, SE, TE, VE) = (G, SI, TI, VI)$$

- Remove the paragraph on interface basis and the extended definition of Inter on a basis.

  > The object classes FunctorClosure and SigEnv describe functors and signature environments, respectively, and are defined in Section 7.2.

- Adapt the definition of the cut down operator $\downarrow$ on environments as follows:

$$(G, SE, TE, VE) \downarrow (G', SI, TI, VI) = (G, SE \downarrow SI, TE \downarrow TI, VE \downarrow VI)$$

  and add the following sentence directly after it:

  > The static semantics ensures that G and G' are equivalent signature environments.

Section 7.3 (Inference Rules):

- Add the following rule for structure declarations:

$$\frac{\text{Inter } B \vdash sigbind \Rightarrow G}{B \vdash \texttt{signature}\ sigbind \Rightarrow G \text{ in Env}} \tag{157a}$$

- In rules 162–175 and 181, except for the ones mentioned in the following, replace all occurrences of IB with I, likewise in the respective boxes giving their form; in those rules already containing occurences of I (162, 165, 173, 181), replace these occurences with I'.

- Change rule 163 as follows:

$$\frac{I(longsigid) = I'}{I \vdash longsigid \Rightarrow I'} \tag{163}$$

- Remove rule 164.
- Add the following rule for specifications:

$$\frac{I \vdash sigdesc \Rightarrow G}{I \vdash \texttt{signature}\ sigdesc \Rightarrow G \text{ in Inter}} \tag{172a}$$

- Add a section for signature description rules of the form

$$\boxed{I \vdash sigdesc \Rightarrow G}$$

  and the following rule:

$$\frac{I \vdash sigexp \Rightarrow I' \qquad \langle I \vdash sigdesc \Rightarrow G \rangle}{I \vdash sigid\ \texttt{=}\ sigexp\ \langle \texttt{and}\ sigdesc \rangle \Rightarrow \{sigid \mapsto I'\}\langle +G \rangle} \tag{181a}$$

- Simplify rule 184 as follows:

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash strdec \Rightarrow E \text{ in Basis}} \tag{184}$$

- Remove rule 185.

Appendix A (Derived Forms):

- In Figure 19, in the box for specifications $spec$, replace the entry for `include` with:

`include` $longsigid_1 \cdots longsigid_n$	`include` $longsigid_1$ `;` $\cdots$ `;` `include` $longsigid_n$

Appendix B (Full Grammar):

- In Figure 23b (as defined in Appendix B.23), replace the $sigexp$ production for signature identifiers to:

$$[sigexp ::=] \qquad longsigid \qquad \text{signature identifier}$$

- In Figure 23c (as defined in Appendix B.23), add the following production for specifications:

$$[spec ::=] \qquad \texttt{signature}\ sigdesc \qquad \text{signature}$$

  and replace the one for multiple include with:

$$[spec ::=] \qquad \texttt{include}\ longsigid_1 \cdots longsigid_n \qquad \text{multiple include}$$

- Add the following production for signature descriptions:

$$sigdesc \quad ::= \quad sigid\ \texttt{=}\ sigexp\ \langle \texttt{and}\ sigdesc \rangle$$

- In Figure 23d (as defined in Appendix B.23), add the following production for structure-level declarations $strdec$:

$$[strdec ::=] \qquad \texttt{signature}\ sigbind \qquad \text{signature}$$

- Remove the production for signature declarations $sigdec$.

- Simplify the definition of $topdec$ to:

$$topdec \quad ::= \quad strdec$$

Appendix C (The Initial Static Basis):

- In the text, replace the definition of B_0 with "$B_0 = T_0, E_0$" and drop "$G_0 = \{\}$ and".
- Replace "$E_0 = (SE_0, TE_0, VE_0)$, where $SE_0 = \{\}$" with "$E_0 = (G_0, SE_0, TE_0, VE_0)$, where $G_0 = \{\}$ and $SE_0 = \{\}$".

Appendix D (The Initial Dynamic Basis):

- Replace the second sentence with

  > The initial dynamic basis is $B_0 = E_0 = (G_0, SE_0, TE_0, VE_0)$, where $G_0 = \{\}$, $SE_0 = \{\}$, TE_0 is shown in Figure 26 and

Compatibility

This is a conservative extension.

B.25 Local Modules

Structure, functor and signature declarations are allowed in local scope:

```
fun sortWithoutDups compare =
    let
        structure Set = MkSet(type t = string
                              val compare = compare)
    in
        Set.toList o foldr Set.insert Set.empty
    end
```

Furthermore, as a derived form, open declarations may contain arbitrary module expressions:

```
fun sortWithoutDups compare =
    let
        open MkSet(type t = string; val compare = compare)
    in
        toList o foldr insert empty
    end
```

Changes to the Definition

The changes described here are relative to the changes for higher-order functors and nested signatures given in Appendices B.23 and B.24.

Section 2.8 (Grammar):

- Extend the first paragraph with the following sentence:

  > In Figure 4, the variable $strdec$ appearing in Figure 4 ranges over the set StrDec of structure-level declarations, which is defined in Section 3.4.

- In Figure 4, add the following production for declarations dec:

  $[dec ::=]$ $\quad strdec \quad$ module declaration

Section 3.3 (Infixed operators):

- In the first paragraph, replace "structure-level declaration $strdec$" with "declaration dec".
- In the list of phrases, replace "$strdec$" with "dec" and remove the phrase concerning `local`.

Section 3.4 (Grammar for Modules):

- In Figure 6, replace all occurrences "$strdec$" in the productions for structure expressions $strexp$ with "dec".
- Remove the productions for core, local, empty and sequential structure-level declarations and the respective restriction note that was added as part of the changes described in Appendix B.1.
- In Figure 8, replace occurrences "$strdec$" in the productions for top-level declarations $topdec$ and the respective restriction note with "dec".

Section 4.10 (Inference Rules):

- Add the following rule for declarations:

$$\frac{C \vdash strdec \Rightarrow E}{C \vdash strdec \Rightarrow E} \tag{20a}$$

  and an accompanying comment:

  (20a) The premise of this rule is a sentence of the static semantics for Modules, see Section 5.7.

Section 5.1 (Semantic Objects):

- In the third paragraph, add the following after the first sentence:

  > Inversely, we define C in Basis to be the basis $(T$ of C, E of $C)$.

Section 5.7 (Inference Rules):

- Change the box giving the form of inference rules for structure expressions to:

$$\boxed{C \vdash strexp \Rightarrow M}$$

  and replace all occurences of B with C in rules 50–55, except for the premises regarding a signature expression $sigexp$ in rules 52, 53 and 55a, where it is replaced by "C in Basis".

- In rules 50 and 55, replace $strdec$ with dec.
- Change the box giving the form of inference rules for structure declarations to:

  $$\boxed{C \vdash strdec \Rightarrow E}$$

  and replace all occurences of B with C in rules 57 and 57a, except for the premise of rule 57a, where it is replaced by "C in Basis".
- Remove rules 56 and 58–60.
- Change the box giving the form of inference rules for structure bindings to:

  $$\boxed{C \vdash strbind \Rightarrow SE}$$

  and replace all occurences of B with C in rule 61.
- In rule 87, replace $strdec$ with dec, and the B in the premise with "C of B".

Section 6.7 (Inference Rules):

- Add the following rule for declarations:

  $$\frac{E \text{ in Basis} \vdash strdec \Rightarrow E'}{E \vdash strdec \Rightarrow E'} \quad (119a)$$

  and an accompanying comment:

  (119a) The premise of this rule is a sentence of the dynamic semantics for Modules, see Section 7.3. The definition of dynamic basis Basis appears in Section 7.2.

Section 7.3 (Inference Rules):

- In rules 150 and 155, replace $strdec$ with dec, and B in the premises with "E of B".
- Remove rules 156 and 158–160.
- In rule 184, replace $strdec$ with dec, and the B in the premise with "E of B".

Appendix A (Derived Forms):

- In Figure 17, add the following to the box for declarations:

`open` *strexp*	`local structure` *strid* `=` *strexp* `in open` *strid* `end`

  and extend the note with

  > [...] and *strexp*; *strid* new

- Add the following bullet to the list of notes regarding Figure 17:

  > In the form involving `open`, the structure expression *strexp* may not be a functor application of the form $longstrid_0\ longstrid_1\ \cdots\ longstrid_n$.

- In Figure 18, replace occurrences of "*strdec*" in the box for structure expressions *strexp* with "*dec*".

Appendix B (Full Grammar):

- In Figure 20, add the following productions for declarations dec:

$[dec ::=]$	$strdec$	module declaration
	`open` $strexp$	open declaration

- In Figure 23a (as defined by Appendices B.23 and B.24), replace all occurrences of "$strdec$" with "dec".
- In Figure 23d (as defined by Appendices B.23 and B.24), remove the productions for core, local, empty and sequential structure-level declarations.
- Replace "$strdec$" in the productions for top-level declarations $topdec$ with "dec".

Compatibility

This is a conservative extension. The syntactic restriction on generalised `open` declarations prevents overlap with the existing form, although deprecation of multiple open might arguably be a preferable solution.

B.26 First-class Modules

Modules can be wrapped up as first-class values, by giving a module expression and an appropriate signature:

```
val p = pack Int : INTEGER
```

The type of such a value is

```
val p : pack INTEGER
```

To unwrap a package, another signature constraint is necessary, e.g.:

```
fun four x =
    let
        structure I = unpack x : INTEGER
    in
        I.toString(I.fromString "4")
    end
```

More extensive examples can be found in `doc/examples/first-class-modules.sml`.

Changes to the Definition

The changes described here are relative to the changes for higher-order and local modules given in Appendices B.23–B.25.

Section 2.1 (Reserved Words):

- Add `pack` to the list of reserved words.

Section 2.3 (Grammar):

- Extend the first paragraph further with the following sentence:

  > Moreover, the variable $longsigid$ occurring in Figures 3 and 4 ranges over the class of long signature identifiers, defined in Section 3.2.

- In Figure 3, add the following production for types:

  $[ty ::=]$ `pack` $longsigid$ first-class module

- In Figure 4, add the following production for expressions:

  $[exp ::=]$ `pack` $longstrid$ `:` $longsigid$ pack module

Section 3.1 (Reserved Words):

- Add `unpack` to the list of reserved words used in Modules.

Section 3.4 (Grammar for Modules):

- In Figure 6, add the following production for structure expressions:

  $[strexp ::=]$ `unpack` $atexp$ `:` $sigexp$ unpack module

Section 4.2 (Compound Objects):

- In Figure 10, extend the definition of Type with "$\cup$ PackType" and add the following:

  $[\Sigma] \quad \in \quad \text{PackType} = \text{Sig}$

- In the paragraph referring to Figure 10, modify the sentence added by the changes described in Appendix B.23 and B.24 to

  > The object classes Sig, FunSig and SigEnv belong to Modules and are defined in Section 5.1.

Section 4.4 (Types and Type functions):

- Add the following bullet to the list of forms that admit equality:
  - $[\Sigma]$, where $\Sigma \in$ Sig.

Section 4.10 (Inference Rules):

- Add the following rule for expressions:

$$\frac{C \text{ in Basis} \vdash longstrid \texttt{ : } longsigid \Rightarrow M \qquad C(longsigid) = \Sigma}{C \vdash \texttt{pack } longstrid \texttt{ : } longsigid \Rightarrow [\Sigma]} \tag{9a}$$

  and an accompanying note:

  (9a) The premise of this rule is a sentence of the static semantics for Modules, see Section 5.7. It ensures that $C(longstrid)$ matches Σ.

- Add the following rule for types:

$$\frac{C(longsigid) = \Sigma}{C \vdash \texttt{pack } longsigid \Rightarrow [\Sigma]} \tag{47a}$$

Section 5.7 (Inference Rules):

- Add the following rule for structure expressions:

$$\frac{C \vdash atexp \Rightarrow [\Sigma] \qquad C \text{ in Basis} \vdash sigexp \Rightarrow (T)M \qquad \Sigma \succsim (T)M \qquad T \cap (T \text{ of } C) = \emptyset}{C \vdash \texttt{unpack}\ atexp\ \texttt{:}\ sigexp \Rightarrow M} \tag{53a}$$

Section 6.3 (Compound Objects):

- In Figure 13, extend the definition of Val with "$\cup$ Mod".

Section 6.7 (Inference Rules):

- Add the following rule for expressions:

$$\frac{E \text{ in Basis} \vdash longstrid\ \texttt{:}\ longsigid \Rightarrow M}{E \vdash \texttt{pack}\ longstrid\ \texttt{:}\ longsigid \Rightarrow M} \tag{103a}$$

  and an accompanying note:

(103a) The premise of this rule is a sentence of the dynamic semantics for Modules, see Section 7.3.

Section 7.3 (Inference Rules):

- Add the following rule for structure expressions:

$$\frac{E \text{ of } B \vdash atexp \Rightarrow M}{B \vdash \texttt{unpack}\ atexp\ \texttt{:}\ sigexp \Rightarrow M} \tag{153a}$$

  and note:

(153a) Because there is no subtyping on package types, the static semantics ensures that M is already cut down to the signature denoted by $sigexp$.

Appendix A (Derived Forms):

- In Figure 15, add the following to the box for expressions:

`pack` $atstrexp$ `:` $atsigexp$	`let structure` $strid$ `=` $atstrexp$ `signature` $sigid$ `=` $atsigexp$ `in pack` $strid$ `:` $sigid$ `end`

($strid$, $sigid$ new)

Appendix B (Full Grammar):

- In Figure 20, add the following production for expressions:

$$[exp ::=] \qquad \texttt{pack}\ atstrexp\ \texttt{:}\ atsigexp \qquad \text{pack module}$$

- In Figure 23, add the following production for types:

$$[ty ::=] \qquad \texttt{pack}\ longsigid \qquad \text{first-class module}$$

- In Figure 23a (as defined by Appendices B.23 and B.24), add the following production for structure expressions:

$$[strexp ::=] \qquad \texttt{unpack}\ atexp\ \texttt{:}\ sigexp \qquad \text{unpack module}$$

Compatibility

Except for the new reserved words `pack` and `unpack` this is a conservative extension.

C Syntax Summary

The following gives a summary of the full grammar as defined by all the changes given in Appendix B. A bullet • marks phrases that are new, a parenthesised bullet (•) phrases that have been extended relative to SML'97.

C.1 Core Language

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvid*	value identifier
		{ ⟨*atexp* `where`⟩ ⟨*exprow*⟩ }	record (•)
		`#` *lab*	record selector
		`(` `)`	0-tuple
		`(` exp_1 `,` ⋯ `,` exp_n `)`	n-tuple, $n \geq 2$
		`[` exp_1 `,` ⋯ `,` exp_n `]`	list, $n \geq 0$
		`(` exp_1 `;` ⋯ `;` exp_n ⟨`;`⟩ `)`	sequence, $n \geq 1$ (•)
		`let` *dec* `in` exp_1 `;` ⋯ `;` exp_n ⟨`;`⟩ `end`	local declaration, $n \geq 1$ (•)
exprow	::=	`...` `=` *exp* ⟨`,` *exprow*⟩	ellipses •
		lab `=` *exp* ⟨`,` *exprow*⟩	expression row
		vid ⟨`:` *ty*⟩ ⟨`,` *exprow*⟩	label as variable •
appexp	::=	*atexp*	
		appexp atexp	application
infexp	::=	*appexp*	
		$infexp_1$ *vid* $infexp_2$	infix application
exp	::=	*infexp*	
		exp `:` *ty*	type constraint (L)
		`pack` *atstrexp* `:` *atsigexp*	pack module •
		exp_1 `andalso` exp_2	conjunction
		exp_1 `orelse` exp_2	disjunction
		exp `handle` ⟨`|`⟩ *match*	handle exception (•)
		`raise` *exp*	raise exception
		`if` exp_1 `then` exp_2 ⟨`else` exp_3⟩	conditional (•)
		`while` exp_1 `do` exp_2	iteration
		`case` *exp* `of` ⟨`|`⟩ *match*	case analysis (•)
		`fn` ⟨`|`⟩ *match*	function (•)

atpat	::=	`_`	wildcard
		scon	special constant
		⟨`op`⟩*longvid*	value identifier
		`?` *atexp*	transformation •
		`{` ⟨*patrow*⟩ `}`	record
		`(` `)`	0-tuple
		`(` pat_1 `,` $\cdots$ `,` pat_n `)`	n-tuple, $n \geq 2$
		`[` pat_1 `,` $\cdots$ `,` pat_n `]`	list, $n \geq 0$
		`(` *pat* `)`	
patrow	::=	`...` ⟨`=` *pat*⟩ ⟨`,` *patrow*⟩	ellipses (•)
		lab `=` *pat* ⟨`,` *patrow*⟩	pattern row
		vid ⟨`:` *ty*⟩ ⟨`as` *pat*⟩ ⟨`,` *patrow*⟩	label as variable
apppat	::=	*atpat*	
		⟨`op`⟩*longvid atpat*	constructed value
		`?` *atexp atpat*	constructed transformation •
infpat	::=	*apppat*	
		$infpat_1$ *vid* $infpat_2$	constructed value (infix)
pat	::=	*infpat*	
		pat `:` *ty*	typed
		pat_1 `as` pat_2	conjunctive (•)
		pat_1 `|` pat_2	disjunctive •
		pat_1 `with` pat_2 `=` *exp*	nested match •
		pat `if` *exp*	guard •
match	::=	*mrule* ⟨`|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	match rule
fmatch	::=	*fmrule* ⟨`|` *fmatch*⟩	
fmrule	::=	*fpat* ⟨`:` *ty*⟩ ⟨`if` *atexp*⟩ `=` *exp*	match clause (•)
fpat	::=	⟨`op`⟩*vid* $atpat_1$ $\cdots$ $atpat_n$	$n \geq 1$
		`(` $atpat_1$ *vid* $atpat_2$ `)` $atpat_3$ $\cdots$ $atpat_n$	$n \geq 3$
		$atpat_1$ *vid* $atpat_2$	

ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record
		tyseq longtycon	type construction
		ty_1 `*` $\cdots$ `*` ty_n	n-tuple, $n \geq 2$
		ty_1 `->` ty_2	function type (R)
		`pack` *longsigid*	first-class module •
		`(` *ty* `)`	
tyrow	::=	`...` `:` *ty* ⟨`,` *tyrow*⟩	ellipses •
		lab `:` *ty* ⟨`,` *tyrow*⟩	type row

dec	::=	`do` *exp*	evaluation •
		`val` ⟨`rec`⟩ *tyvarseq valbind*	value (•)
		`fun` *tyvarseq funbind*	function
		`type` *typbind*	type
		`datatype` *datbind* ⟨`withtype` *typbind*⟩	datatype
		`datatype` *tycon* = `datatype` *longtycon*	replication
		`viewtype` *tyvarseq tycon* = *ty* `as` ⟨`|`⟩ *conbind* `with` *dec* `end`	viewtype •
		`viewtype` *tycon* = `viewtype` *longtycon*	viewtype replication •
		`abstype` *datbind* ⟨`withtype` *typbind*⟩ `with` *dec* `end`	abstract type
		`exception` *exbind*	exception
		strdec	module declaration •
		`open` *strexp*	open •
		`open` $longstrid_1 \cdots longstrid_n$	multiple open
		`local` dec_1 `in` dec_2 `end`	local
			empty
		dec_1 ⟨`;`⟩ dec_2	sequential
		`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix left directive, $n \geq 1$
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix right directive, $n \geq 1$
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive, $n \geq 1$
valbind	::=	*pat* = *exp* ⟨`and` *valbind*⟩	
fvalbind	::=	⟨`|`⟩ *fmatch* ⟨`and` *fvalbind*⟩	(•)
typdesc	::=	*tyvarseq tycon* = *ty* ⟨`and` *typbind*⟩	
datbind	::=	*tyvarseq tycon* = ⟨`|`⟩ *conbind* ⟨`and` *datbind*⟩	(•)
conbind	::=	⟨`op`⟩*vid* ⟨`of` *ty*⟩ ⟨`|` *conbind*⟩	
exbind	::=	⟨`op`⟩*vid* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		⟨`op`⟩*vid* = ⟨`op`⟩*longvid* ⟨`and` *exbind*⟩	

C.2 Module Language

atstrexp	::=	`struct` *dec* `end`	basic
		`(` *dec* `)`	basic (short) •
		longstrid	structure identifier
		`let` *dec* `in` *strexp* `end`	local declaration
		`(` *strexp* `)`	•
appstrexp	::=	*atstrexp*	
		appstrexp atstrexp	functor application (•)
strexp	::=	*appstrexp*	
		strexp `:` *sigexp*	transparent constraint
		strexp `:>` *sigexp*	opaque constraint
		`unpack` *atexp* `:` *sigexp*	unpack module •
		`fct` *strid* `:` *sigexp* `=>` *strexp*	functor •
		`fct (` *spec* `) =>` *strexp*	functor (short) •

atsigexp	::=	`sig` *spec* `end`	basic
		`(` *spec* `)`	basic (short) •
		longsigid	signature identifier (•)
		`(` *sigexp* `)`	•
sigexp	::=	*atsigexp*	
		sigexp `where type` *tyvarseq longtycon* `=` *ty*	type realisation
		`fct` *strid* `:` *atsigexp* `->` *sigexp*	functor •
		`fct (` *spec* `) ->` *sigexp*	functor (short) •
		atsigexp `->` *sigexp*	non-dependent functor •

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`type` *syndesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc* ⟨`withtype` *typbind*⟩	datatype (•)
		`datatype` *tycon* `=` `datatype` *longtycon*	replication
		`viewtype` *tyvarseq tycon* `=` *ty* `as` ⟨`|`⟩ *condesc*	viewtype •
		`viewtype` *tycon* `=` `viewtype` *longtycon*	viewtype replication •
		`exception` *exdesc*	exception
		`structure` *strdesc*	structure
		`functor` *fundesc*	functor •
		`signature` *sigdesc*	signature •
		`include` *sigexp*	include
		`include` $longsigid_1 \cdots longsigid_n$	multiple include (•)
			empty
		$spec_1$ ⟨`;`⟩ $spec_2$	sequential
		spec `sharing type` $longtycon_1 = \cdots = longtycon_n$	type sharing $(n \geq 2)$
		spec `sharing` $longstrid_1 = \cdots = longstrid_n$	structure sharing $(n \geq 2)$
valdesc	::=	*vid* `:` *ty* ⟨`and` *valdesc*⟩	
typdesc	::=	*tyvarseq tycon* ⟨`and` *typdesc*⟩	
syndesc	::=	*tyvarseq tycon* `=` *ty* ⟨`and` *syndesc*⟩	
datdesc	::=	*tyvarseq tycon* `=` ⟨`|`⟩ *condesc* ⟨`and` *datdesc*⟩	(•)
condesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`|` *condesc*⟩	
exdesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`and` *exdesc*⟩	
strdesc	::=	*strid* `:` *sigexp* ⟨`and` *strdesc*⟩	
fundesc	::=	*strid* `(` $funarg_1$ `)` $\cdots$ `(` $funarg_n$ `)` `:` *sigexp* ⟨`and` *fundesc*⟩	$(n \geq 1)$ •
funarg	::=	*strid* `:` *sigexp*	
		spec	
sigdesc	::=	*sigid* `=` *sigexp* ⟨`and` *sigdesc*⟩	•

strdec	::=	`structure` *strbind*	structure
		`functor` *funbind*	functor ($\bullet$)
		`signature` *sigbind*	signature ($\bullet$)
strbind	::=	*strid* $\langle$`:`$\langle$`>`$\rangle$ *sigexp*$\rangle$ `=` *strexp* $\langle$`and` *strbind*$\rangle$	
funbind	::=	*strid* `(` $funarg_1$ `)` $\cdots$ `(` $funarg_n$ `)`	($n \geq 1$) ($\bullet$)
		$\langle$`:`$\langle$`>`$\rangle$ *sigexp*$\rangle$ `=` *strexp* $\langle$`and` *funbind*$\rangle$	
sigbind	::=	*sigid* `=` *sigexp* $\langle$`and` *sigbind*$\rangle$	
topdec	::=	*dec*	($\bullet$)
program	::=	*topdec* `;` $\langle$*program*$\rangle$	
		exp `;` $\langle$*program*$\rangle$	

D 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 ↓ 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 1.2.2/S1 (2005/12/12)

Interpreter:

- Implemented RFC: Syntax fixes.
- Implemented RFC: Semantic fixes.
- Implemented RFC: Line comments.
- Implemented RFC: Extended literal syntax.
- Implemented RFC: Record punning.
- Implemented RFC: Record extension.
- Implemented RFC: Record update.
- Implemented RFC: Disjunctive patterns.
- Implemented RFC: Conjunctive patterns.
- Implemented RFC: Match guards.
- Implemented RFC: Optional bar in matches.
- Implemented RFC: Simplified recursive bindings.
- Implemented RFC: Strengthened value restriction.
- Implemented RFC: Degraded abstype.
- Implemented RFC: Proper scoping for transparent type specifications.
- Implemented RFC: Withtype specifications.
- Implemented RFC: Remove ”and” in type realisations.

Version 1.2.2/S2 (2006/01/02)

Interpreter:

- Implemented RFC: Do declarations.
- Extended RFC: Record extension to support record type extension and freely placed ellipses.
- Fixed bug in record type field lookup.

Version 1.2.3/S2 (2006/07/18)

Merged changes from 1.2.3.

Version 1.2.4/S2 (2006/08/14)

Documentation:

- Clarified license.

Version 1.2.4/S3 (2006/09/10)

Interpreter:

- Modified RFC: Line comments to use `(*)` as delimiter.
- Extended RFC: Optional bar in matches to support datatype declarations and specifications.

Version 1.3.0/S4 (2007/03/22)

Merged changes from 1.3.0, plus:

Interpreter:

- Implemented RFC: Views.
- Implemented RFC: Nested matches.
- Implemented RFC: Transformation patterns.
- Generalised RFC: Match guards to Pattern guards.
- Implemented RFC: Higher-order functors.
- Implemented RFC: Nested signatures.
- Implemented RFC: Local modules.
- Implemented RFC: First-class modules.
- Extended RFC: Optional bars to cover semicolons as well.

Documentation:

- Added Appendix B documenting all extensions.

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 1.3.1/S5 (2008/04/28)

Merged changes from 1.3.1, plus:

Interpreter:

- Implemented RFC: Optional `else` branch.
- Fixed and simplified definition of signature matching for RFC: Higher-order functors.

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/`

[SML05] *Successor ML*
`http://www.successor-ml.org/`

[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`

[MT94] Dave MacQueen, Mads Tofte
A Semantics for Higher-order Functors
in: Proc. of the 5th European Symposium on Programming
Springer-Verlag, 1994

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

[L97] Mark Lillibridge
Translucent Sums: A Foundation for Higher-Order Module Systems
PhD Thesis
School of Computer Science, Carnegie Mellon University, 1997

[W87] Philip Wadler
Views: a way for pattern matching to cohabit with data abstraction
in: Proc. of the 14th Annual ACM Symposium on Principles of Programming Languages
ACM Press, 1987

[O98] Chris Okasaki
Views for Standard ML
in: 1998 ACM SIGPLAN Workshop on ML
ACM Press, 1998

[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 ACM 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://cm.bell-labs.com/cm/cs/what/smlnj/`

[NJ98] *The SML/NJ Library*
`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/smlnj-lib/index.html`

[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 2.01)
`http://www.dina.kvl.dk/ sestoft/mosml.html`

[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`