MLj 0.2 User Guide

Nick Benton, Andrew Kennedy, George Russell
Persimmon IT, Inc.
Cambridge, U.K.

February 28, 1999

Contents

1 Introduction

The MLj Compiler is a complete system for Standard ML to Java[1] bytecode compilation. MLj can be operated from the command line or interactively from a compilation environment. It manages multiple modules automatically; parsing, type checking, and partially compiling separate SML structures and signatures as necessary and then linking them together, applying various optimisations and finally producing a single `zip` file containing Java classes. This approach to separate compilation is unusual. No intermediate object files are produced (say, as separate Java classes) and most of the work in compilation happens *after* linking. Of course, this has an adverse effect on the speed of re-compilation, but it does enable better code to be produced.

1.1 What does it compile?

The current version of MLj compiles a subset of SML '97 [6], approximately everything except for functors. There are a few other minor differences documented in Section 6. It is important to note that MLj compiles only stand-alone modular applications: the `use` command is not available and programs consist of a collection of top-level structures and signatures. In particular there is no interactive read-eval-print loop; the online example in `demos/hal` illustrates one way of overcoming this restriction.

Almost all of the Standard ML Basis Library [1] is implemented. Omissions and discrepancies are described in Section 8.

Finally, a number of language extensions are provided for interfacing existing Java classes and for implementing new classes with methods written in SML. These are discussed in full in Section 7.

1.2 About this document

This guide is aimed at programmers already familiar with SML but not necessarily with any knowledge of Java. The textbook by Paulson [7] is an up-to-date introduction to SML'97.

Section 2 describes the installation procedure. Then Section 3 introduces the compilation environment by leading you through the steps required to compile a single-module text-only application. Techniques required to compile multi-module programs are discussed in Section 4. Section 5 is a complete reference to the compiler command language. Section 6 describes the current omissions from the full Standard ML language. Section 7 describes extensions to Standard

[1] Java is a trademark of Sun Microsystems

ML for interfacing to Java class libraries and for implementing new classes inside ML. Section 8 lists the elements of the Standard Basis Library that are implemented and any omissions and discrepancies.

1.3 Changes for version 0.2

Version 0.2 is a snapshot of internal development at Persimmon IT up to February 1999. As such, it has many improvements over the original release but some changes are not yet complete or may not be robust.

As far as users of MLJ 0.1 are concerned, the major changes are to the Java extensions to ML. Some features have been superceded and are flagged as "deprecated" to encourage users to switch to the new syntax. These are the following: all types and coercions in the structure `Java` (no longer necessary), quoted Java class types (the quotes can be omitted), static field and method access (see described in Section 7.3 for the new approach), non-static field and method access (see Section 7.4), static field and method declarations (use the new structure export mechanism), and casts (see Section 7.4.3).

The `_classtype` construct remains but is likely to change substantially in future releases, to integrate Java and ML more closely.

The compilation environment remains much the same, except for an extension to the `export` and `make` commands to permit structure exports, and a new `bootclasspath` command and associated `BOOTCLASSPATH` environment variable.

1.4 Copyright notice for MLj

1.5 Acknowledgements

MLjwas developed by Nick Benton, Andrew Kennedy and George Russell at Persimmon IT Inc. and initially released in May 1998. Development was discontinued in February 1999 and a snapshot of development released open-source under the the Gnu Public License. MLj is available for download from a number of sites and the original authors continue to take an interest in its development.

I would like to thank Dave Halls and Audrey Tan of Persimmon IT for their help as users of the compiler. Outside Persimmon, I am particularly grateful to Ian Stark, Bent Thomsen, Lone Thomsen, and Stephen Gilmore for their suggestions and support.

Andrew Kennedy, February 1999.

2 Installation

2.1 Components of an MLj system

The Standard ML of New Jersey compiler. The MLj compiler was developed using SML of New Jersey. At present MLj is supplied as source files only; therefore you need SML/NJ version 110 or above to build an appropriate heap image.

The MLj compiler. Build instructions can be found in the `INSTALL` file; once built, MLj requires only the SML/NJ run-time system.

An editor. The MLj compiler is not an integrated development environment, so you will need a text editor which typically you will run concurrently with the MLj compilation environment.

A Java standard class library. Java application programs are run using a bytecode interpreter such as `java` (provided by Sun and third parties) or Microsoft's `jview`. Java applets are run from web browsers. Both require a collection of standard classes, some of which form part of the Java language specification (with names `java.lang.*`) and others providing a variety of services (such as input/output in `java.io.*` and GUI support in `java.awt.*`).

The MLj compiler needs access to these classes in order to type check and compile code making use of Java extensions to ML. Because there are so many classes, they are usually packaged up in some way. MLj will process the uncompressed `zip` format directly; for compressed files it requires the `unzip` program.

For the JDK from Sun prior to version 1.2, the standard class library is found in a file called `classes.zip` in a directory `lib`. For JDK 1.2 (also known as "Java 2"), it is found in a file `rt.jar` in a directory `jre/lib`. In the case of Microsoft's SDK for Java, you may need to run `clspack -auto` to create a `classes.zip` file.

either **A Java bytecode interpreter**. This is required if you want to run standalone Java application programs. The interpreter from Sun (which has been ported by third parties) is called `java`, and Microsoft supply one as part of their SDK for Java called `jview`.

Even if you intend to use MLj to write applets only, you may find a standalone bytecode interpreter useful. Through some trickery it is possible to run an applet as an application, though of course any interaction with the web browser cannot be simulated. You can then run the applet directly within MLj's compilation environment.

or **A Java-enabled web browser**. Web browsers run Java *applets*. These are simply instances of the Java class `java.applet.Applet` and can be created easily with MLj.

Unfortunately some browsers (notably Netscape's prior to version 4.0) make it hard for developers: classes are not automatically reloaded when they have been updated through re-compilation. Even clearing the caches doesn't help. So you may find Sun's `appletviewer` program or Microsoft's `jview` useful in this case.

On Netscape Navigator version 4.0 and above, you can use *Shift-Reload* to force reloading of classes. On Microsoft's Internet Explorer, use *Ctrl-Refresh*.

(optionally) **A Java compiler**. In addition to using the Java extensions to ML you can write parts of a program in pure Java which is then called from ML. For this you will require a Java compiler such as Sun's `javac` or Microsoft's `jvc`.

2.2 Choice of platform, JVM and API

The Standard ML Basis Library is pre-compiled into the MLj heap image and comes in two flavours: one that makes use of features found only in the Java API version 1.1.1 and above, and another for previous versions. If you have a bytecode interpreter, you can type `java -version` to find out which version you have. Even if this is up-to-date, you may want to compile applets that can be run from older browsers.

At present, the only parts of the Basis that are implemented differently in the two versions are the `IntInf` and `Date` structures and the functions `Real.fmt` and `Real.fromString`. Programs not making use of these features will produce the same compiled code in both versions. Note that compiled code from programs using these features will be larger under the older API.

The quality of currently available JVM implementations varies widely. Many are buggy – and some, in particular, seem to work well on code compiled from Java but less reliably on code compiled using MLj. The performance of different JVMs is also very variable. Do not despair if MLj programs have poor performance under a particular JVM, as it is likely that they will run much faster under a different JVM – and this situation will improve in the future.

For the Pentium, the most robust JVM is currently Symantec's Just-In-Time compiler that forms part of Sun's JDK 1.2. However, be sure to download the latest update (version 3.10.100), available from the Java Developer Connection, as this fixes a number of major bugs.

Microsoft's just-in-time JVM implementation, built into Internet Explorer and supplied with their Java SDK, is also good. Again, download the latest version (build 3167) from Microsoft's website. There are still bugs but they appear to be confined to floating-point arithmetic (affecting an FFT program, for example).

2.3 Environment

Change your user profile so that the following environment variables are set:

- Extend `PATH` with the `bin` directory containing the `mlj` executables.
- Set `BOOTCLASSPATH` to a path in which the system classes live, as described earlier.

  The system classes are required both by the compiler (to type-check SML calls to Java) and by the bytecode interpreter.
- Set `MLJJVM` to a prefix of the command line that you would type to run the standalone bytecode interpreter. It must include the name of the command and the option used to set the class path.
  - For Sun's JDK, set it to `java -classpath`.
  - For Microsoft's SDK for Java, set it to `jview /q /cp`.
- Set `MLJBIN` to the `bin` directory containing the executables.
- Set `NJBIN` to the `bin` directory containing the SML/NJ run-time system and heap images.
- If any classes in `zip` or `jar` files in the class path are compressed, then set `MLJUNZIP` to a prefix of the command line that you would type to unzip an archive with the following options: preserve case of filenames, display no messages, and pipe the output. If this variable is left unset, it defaults to `unzip -Uqqp`.

3 Single module applications

The compiler is invoked by typing `mlj` at the command line (if you have installed both Java 1.0 and 1.1 versions but want to use Java 1.0 then type `mlj -java10` instead):

```
$mlj
MLj 0.2 on sparc under solaris with basis library for JDK 1.1
Copyright (C) 1999 Persimmon IT Inc.

MLj comes with ABSOLUTELY NO WARRANTY. It is free software, and you are
welcome to redistribute it under certain conditions.
See COPYING for details.
\
```

3.1 Example 1: quicksort

To test its operation on a demonstration program, type the commands that follow the \ prompts below:

```
\ sourcepath demos/sort
\ make Sort
Checking timestamps on source files......done.
Analysing dependencies...
  Parsing structure Sort...done.
...done.
Type checking structure Sort...done.
Compiling structure Sort..........done.
Linking modules...done.
Compiling whole program.............................................done.
\ run 30
Before sorting: 92 74 79 60 14 10 27 53 27 95 77 9 56 13 32 52 84 53 4
31 86 58 59 11 45 40 68 99 27 35
After sorting: 4 9 10 11 13 14 27 27 27 31 32 35 40 45 52 53 53 56 58
59 60 68 74 77 79 84 86 92 95 99
\ quit
$
```

The `sourcepath` command tells the compiler where to look for SML source files, which by default have the extensions `.sml` (for structures) and `.sig` (for signatures). Note that under Windows, the '/' character is automatically translated into the more usual '\'. Then `make` tells MLj to compile and link a program whose single *exported* Java class is defined by an ML structure `Sort`, by default given the class name `Sort`. The compiler will put the output (a collection of Java classes including `Sort`) in a file `Sort.zip`. Finally `run` executes the program with the arguments specified, assuming that the `MLJJVM` environment variable has been set up appropriately. The `help` command lists concisely the syntax of all MLj commands. For more detail on a particular command you can type `help` *command*.

Just to prove that we really have compiled a self-contained Java program, from an OS command prompt type

```
$ java -classpath Sort.zip Sort 20
Before sorting: 77 9 56 13 32 52 84 53 4 31 86 58 59 11 45 40 68 99 27 35
After sorting: 4 9 11 13 27 31 32 35 40 45 52 53 56 58 59 68 77 84 86 99
$
```

3.2 Exporting classes

If you have `unzip` installed), you can type `"unzip -lU Sort.zip"` to peek at the contents of the output file. (Any operating system command can be invoked from the compilation environment by enclosing it in quotes). You will see something like

```
Archive:  Sort.zip
```

```
 Length     Date   Time    Name
--------    ----   ----    ----
    1275  02-22-99 18:05   Sort.class
     387  02-22-99 18:05   Fa.class
     186  02-22-99 18:05   F.class
     387  02-22-99 18:05   E.class
     276  02-22-99 18:05   Ea.class
     243  02-22-99 18:05   Eb.class
    1976  02-22-99 18:05   G.class
     334  02-22-99 18:05   Re.class
     183  02-22-99 18:05   Rd.class
     215  02-22-99 18:05   Rc.class
     211  02-22-99 18:05   Rb.class
     221  02-22-99 18:05   Ra.class
--------                   -------
    5894                   12 files
```

Most of the classes were generated by MLj to implement ML code, but `Sort` is special. The command `make Sort` is actually shorthand for `export Sort` followed by `make`. At its simplest, `export` is followed by a comma-separated list of top-level SML structures, each of which will be *exported* as a Java class with the same name. The signature of each structure determines what will appear in the class. Functions are exported as static Java methods, and other values are exported as static final Java fields. There are strong restrictions on the types of functions and values that can be exported, described in detail in Section 7.

The example `Sort` structure has the following signature:

```
sig
  val main : string option array option -> unit
end
```

The resulting class has a method `main` with void return type and single argument of type `java.lang.String[]`. This is the pattern required by Java interpreters for standalone programs.

3.3 Writing your own text application

You can follow the model set by `Sort` to write your own text applications, without using any of the Java extensions to ML provided by MLj. Simply write a top-level structure with the signature given above, and export it as a Java class. To ease access to command-line arguments, you can use a special Basis function `CommandLine.init` as illustrated below:

```
fun main a = (CommandLine.init a; ...)
```

The arguments can then be accessed using `CommandLine.arguments`.

4 Multiple module applications

4.1 Example 2: drawing trees

Our second example demonstrates something that is rather hard to do with typical SML implementations: graphics. Follow the sequence of commands shown below:

```
\ sourcepath demos/trees
\ make Main, Applet.TreeApplet
...
Compiling whole program...........................................done.
\ run
```

A window should appear displaying a randomly-generated tree: try clicking on the tree to generate and display a fresh one, then close the window. This demonstration can also be run as a Java applet, either from a browser or using a viewer such as Sun's `appletviewer`. The HTML file `demos/trees/index.html` demonstrates this.

To write your own graphical applications it is necessary to understand in detail the Java extensions to SML. These are described in Section 7.

4.2 Mapping of module identifiers to files

A multiple module MLj program consists of a collection of top-level SML structures and signatures, each stored in a separate file. In contrast to other compilation managers, MLj does not require the programmer to list explicitly the files making up a project. Instead, given the location of SML-defined structures and types to be exported as classes with given names (using `export`) and a means of mapping SML signature and structure identifiers onto file names, MLj determines automatically which files it must compile.

The most straightforward way to operate is to put each structure *strid* into a file called *strid*`.sml` and each signature *sigid* into a file called *sigid*`.sig`. Users of Moscow ML will be familiar with this pattern. The compiler then merely requires a path in which to find these files and a root structure (or structures) from which to begin its dependency analysis. Our example program illustrates this technique. All source files are found in the directory `demos/trees`. A structure called `Applet`, stored in the file `demos/trees/Applet.sml`, and the structure `Main`, stored in the file `demos/trees/Main.sml`, are the two roots.

The structure `Main` refers to a structure `Tree`, stored in file `trees/Tree.sml`, and so on. All that the programmer has to do is conform to this naming convention and specify a path and export list.

The command

`sourcepath` *directory* ... *directory*

specifies a list of directories to search for source files. Note that the directory names are separated by spaces. To display the current path type `sourcepath?`.

4.3 User-defined mapping

Sometimes it may be necessary to put SML entities in files not conforming to the default naming conventions. In this case, you can define your own mapping from structure or signature identifier to filename. The commands

`structurefrom` *strid filename* , ... , *strid filename*

and

`signaturefrom` *sigid filename* , ... , *sigid filename*

let you do this. If a filename includes a directory, then this overrides the specified `sourcepath`. Otherwise, the filename specified will be searched for in the directories listed by `sourcepath`.

4.4 Current directory

Relative path and file names in MLj are interpreted in the directory which is current *at the time they are accessed.* Usually this means at `make` time. You can change the current directory by typing `cd` *directory* and query its current value with `cd?`. So if you set up paths and structure mappings and then change directory, any relative directory or file names will be interpreted relative to this new current directory when `make` is invoked.

4.5 Recompilation

When a source file is changed and `make` is invoked, MLj will recompile that file and, if necessary, any other files that depend on it. This propagation will only happen if the result of *elaborating* the entity has changed (in essence, its *type*). In addition to the usual software engineering concerns, this is one more reason to hide the internals of a module by an SML signature. If a recompiled structure matches an unchanged signature then no modules which depend on it will be recompiled.

At present MLj does not keep intermediate results beyond the end of a session. So if you exit to the command-line and then return to an MLj session, files will be re-parsed, type-checked and partially compiled even if they have not changed. A future release of the compiler will introduce persistence of intermediate results across sessions.

5 Summary of commands

5.1 Command syntax

In general a command consists of a alphabetic keyword optionally followed by a list of arguments separated by spaces. The case of keywords is ignored. Arguments containing spaces or the symbols '`+`', '`,`' or '`?`' must be enclosed in quotes as these symbols are used in the syntax of certain commands.

Many commands change the value of some setting such as a path or filename. For these the current setting can be queried by typing the command followed by '`?`'. For example, `log?` displays the filename currently used for logging compiler messages.

Certain commands set the value of a list which is searched from left to right for files (in the case of `classpath` and `sourcepath`) or some other entity (signature identifiers in `signaturefrom` and structure identifiers in `structurefrom`). All of these commands have the general syntax

command [`+`] *args* [`+`]

The form *command args* simply replaces the current value of the list by *args*. The form *command* `+` *args* appends *args* to the end of the list; hence they will be searched last. The form *command args* `+` prepends *args* to the front of the list; hence they will be searched first.

5.2 Command files and command-line operation

A sequence of compiler commands can be collected together in a file *name*`.mlj` and then executed simply by typing *name* (assuming that *name* does not clash with one of the built-in commands). Inside *name*`.mlj` the commands must be separated by newlines, *except* in the case of `export`, `structurefrom` and `signaturefrom` which can be spread over several lines provided that the splits occur following the '`,`' character which separates bindings.

It is also possible to execute compiler commands directly from the command line, either before entering the compilation environment (to set options such as `classpath`, for example) or without entering the environment at all. The method is simple: just precede the commands with hypens (*à la* Unix command options), ending with `-quit` if you do not want to enter the compilation environment. For example,

```
$ mlj -sourcepath demos/trees -make Main, Applet.TreeApplet -quit
```

will compile our graphical example to produce a file `Main.zip`.

There is one command-line option that is not available from within the environment. If you have installed both Java 1.0 and 1.1 versions, then just typing `mlj` will default to Java 1.1. If the *first* option passed to the compiler is `-java10` then this default is overridden and Java 1.0 used instead.

5.3 Alphabetic listing of commands

`bootclasspath` [`+`] *directory* ... *directory* [`+`]

Set, append to, or prepend to the list of directories, `zip` and `jar` files in which the compiler first looks for classes imported by SML programs. In contrast to `classpath`, this path is *not* passed as an argument to the Java interpreter invoked with `run`.

Upon startup, the MLj compiler sets `bootclasspath` to the value of the environment variable `BOOTCLASSPATH`.

`cd` *directory*
: Change the current directory.

`classpath` [+] *directory* … *directory* [+]
: Set, append to, or prepend to the list of directories, `zip` and `jar` files in which the compiler looks for classes imported by SML programs, *after* it has searched `bootclasspath`. The class path is also passed to the Java interpreter invoked with `run`.

  Upon startup, the MLj compiler sets `classpath` to the value of the environment variable `MLJCLASSPATH`.

`export` [+] *longid* [*classname*] `,` … `,` *longid* [*classname*] [+]
: Specify SML-defined classes that are to be exported with given names.

  If *longid* is a simple structure identifier, then that structure is exported as a class with the name given; if the name is omitted, it defaults to the structure name itself. Each binding listed in the signature of the structure must be *exportable* as defined in the separate manual.

  If *longid* is a fully qualified identifier, it refers to a type defined using the `_classtype` construct, exported as a class with the name given; if the name is omitted it defaults to the type name itself (so `Applet.TreeApplet` is exported as a class `TreeApplet`). All class types listed must be *exportable* as defined in the separate manual.

  The list of classes specified by this command is the only information that the compiler uses to determine which structures and signatures to compile: they form the 'root' for its dependency analysis of your program.

`help`
: List the commands available in the compilation environment.

`help` *command*
: Display a more detailed description of the command specified.

`help signature` *sigid*
: Display the types and values defined in an SML signature in a similar style to the `help structure` command described below.

`help structure` *strid*
: Display the types and values available in a given SML structure. The structure must already have been compiled successfully, either belonging to the Basis Library (in which case it is precompiled and forms part of the MLj system) or accessible from the current project.

  Note that the types of value identifiers displayed cannot be used to form a signature against which the specified structure could be matched (that is, you cannot write *strid* `:` ⟨*result of help*⟩). This anomaly will be fixed in a future version of the compiler.

`help class` *classid*
: Display a `_classtype` signature for the *external* class given.

`help class` *classid* `+`
: Display a `_classtype` signature for the external class given, including any fields or methods that it inherits from its superclass and superinterfaces.

`jvm` *OS-command*
: Set the operating system command that is used by `run` to execute standalone applications inside the compilation environment. It must include the option tag that is used to set the class path. For Sun's JDK it should be set to `java -classpath` is appropriate, and for Microsoft's SDK for Java, `jview /q /cp` can be used. The `run` command appends to this the classes listed in `classpath`, the `zip` file specified by `target`, the name of the class containing a `main` method, and finally the specified arguments. If you want to set other options (for example, to set the stack size) then these must precede the class path tag. For example, setting `jvm` to `java -oss1000000 -classpath` would provide a larger stack.

  Upon startup, the MLj compiler sets `jvm` to the value of the environment variable `MLJJVM` if it exists; otherwise, it defaults to `java -classpath`.

`log` *filename*
: Instruct MLj to copy messages produced by `make` to a file, including additional diagnostic information in the case of a compiler bug.

  The default behaviour is to produce no log.

`log`
: Turn off logging.

`make`
: Parse, type check and compile the current program whose 'root' structures are determined by the `export` command.

`make` *args*
: Shorthand for `export` *args* followed by `make`.

`on` *switch* and `off` *switch*
: Turn a compiler switch on or off. Currently available compiler switches are as follows:

  - `exnlocs`: exception location information (default: off). If enabled, the `make` command will insert appropriate code into the output so that when ML exceptions appear at top level the Java interpreter reports the SML structure and line number where the exception was raised. This assumes that the Java interpreter invokes the `toString` method on uncaught exceptions. If not, you can do it yourself:

```
(your program) handle e =>
(print (valOf(e.#toString ())); raise e)
```

- `valuewarning`: non-generalised type variable warnings (default: on). These are displayed whenever SML's *value restriction* prevents the generalisation of a type in a `val` binding (see Section 6.4 for more details).

`quit`
: End the MLj session.

`run` [*arg* ... *arg*]
: Execute a program that has been compiled successfully.

  If exactly one of the structures exported by the `export` command has an appropriate method `main` then the standalone Java interpreter is run to invoke `main` with the arguments specified.

  Otherwise, the program is not run. A future extension would permit the execution of applets using an applet viewer.

  The command used to execute the Java interpreter can be set using `jvm`.

`signaturefrom` [+] *sigid filename* , ... , *sigid filename* [+]
: Override the default *sigid*`.sig` signature-to-filename mapping for specified signature identifiers.

`sourcepath` [+] *directory* ... *directory* [+]
: Set, append to or prepend to the list of directories in which the compiler looks for SML structures and signatures.

`structurefrom` [+] *strid filename* , ... , *strid filename* [+]
: Override the default *strid*`.sml` structure-to-filename mapping for specified structure identifiers.

`target` *filename*
: Specify where the (zipped) classes output by the compiler output are to be stored.

`target`
: Return `target` to its default setting. Classes are saved in a file *strid*`.zip` where *strid* is the name of the first SML structure listed by `export`.

`"`*OS-command*`"`
: Execute the operating system command enclosed in quotes.

6 Language restrictions

6.1 No functors

The current version of MLj does not implement functors or sharing constraints in signatures. However, all other features of the module system are available (substructures, the `where` construct new to SML '97, and so on). Functors will be added in a future release.

6.2 Overflow

The Standard ML Basis library requires certain arithmetic operations to raise an `Overflow` exception when the result is not representable (*e.g.* `Int.+`, `Int.*`); the same operations in Java wrap around without raising an exception. A correct implementation in Java bytecodes of these Basis operations would have a performance unacceptable in most applications, so it was decided to diverge from the standard and to raise no exception. If there is sufficient interest, a future release may include a special version of the Basis in which `Overflow` *is* raised. Even so, this would probably best be used to track down bugs (for instance, turning an infinite loop into an uncaught exception) and not relied on for production code.

6.3 Non-uniform datatypes

The MLj compiler imposes the restriction that occurrences of parameterised datatypes within their own definition are applied to the same type arguments as the definition. In any case datatypes such as

```
datatype 'a Weird = Empty | Weird of ('a*'a) Weird
```

are of limited use in the absence of polymorphic recursion.

MLj also insists that datatype and exception declarations contain no free type variables (as with `fn x:'a => let exception E of 'a in ...`).

These restrictions will be lifted in a future release of the compiler.

6.4 Value restriction

The definition of SML '97 specifies that the types of variables in bindings of the form

`val` *pat* `=` *exp*

are generalised to allow polymorphism *only* when *exp* is a syntactic value (*non-expansive expression* [6, Section 4.7]). MLj makes the further restriction that generalisation can only occur if *pat* is *non-refutable*, that is, a match will always succeed and not raise the `Bind` exception. (An example of a refutable binding is `val [x] = nil::nil`). This restriction is also applied by SML/NJ version 110 and it can be argued that it is an omission from the Definition (indeed, the restriction is included in a recent type-theoretic recasting of the semantics of SML '97 [5]).

MLj also prevents generalisation when *pat* contains `ref` patterns. This second restriction will be lifted in a future release.

6.5 Overloading

MLj implements overloading of constants and operators for all the types required by the Basis Definition, except that integer constants cannot have type

`LargeInt.int`. Use `LargeInt.fromInt` or `LargeInt.fromString` to construct values of this type. This omission will be removed in a future release.

MLj resolves default types for overloaded constants and operators at each `val` or `fun` binding. This is a smaller context than that used by other implementations but is permitted by the Definition [6, Appendix E]. The following typechecks under SML/NJ and Moscow ML but not under MLj because `x` is assumed to have the default type `int` at the binding of `sqr`.

```
fun g (x,y) =
let
  fun sqr x = x*x
in
  sqr (x+2.0) + y
end
```

For maximum compatibility with other implementations a future version of MLj will use the largest context permitted.

6.6 Tail call behaviour

Although not part of the Definition of Standard ML, it is commonly assumed by functional programmers that the implementation of a tail call (where the result of one function is given by a call to another function) will re-use the stack frame of the calling function for the callee. In particular, a purely tail-calling recursive function can consume constant store.

MLj attempts to inline function calls or to translate them into `goto` bytecodes where it can. However, for the remaining cases there is no obligation on the part of the bytecode interpreter to implement tail calls properly (though it is explicitly permitted in the Java language specification [2, §15.11.4.6]). Most current interpreters and JIT compilers do not optimise tail calls.[2] (The Microsoft JIT optimises single-recursion static tail calls but these can be implemented using `goto` anyway). If this situation persists then it will be necessary to change the code output by MLj in this respect.

[2] The reason this is not *completely* straightforward for Java is that some SecurityManagers make use of the call chain. Tail call elimination is only allowable (roughly) between classes having the same class loader.

ML type	Java type
`int`	`int`
`real`	`double`
`char`	`char`
`bool`	`boolean`
`string`	`java.lang.String`
`exn`	`java.lang.Exception`
`array`	`[]`
`Int8.int`	`byte`
`Int16.int`	`short`
`Int64.int`	`long`
`Real32.real`	`float`
`IntInf.int`	`java.math.BigInteger`
`Date.date`	`java.util.Calendar`

Table 1: ML and Java types

7 Language extensions

In this section we discuss the extensions to the Standard ML language that facilitate access to Java libraries and allow the creation of new libraries written using ML.

7.1 Design rationale

The main principle is *simplicity*. The syntax used to access Java classes and to create new ones has been designed to be as lightweight as possible so that one can use it without even thinking "foreign call". To attain simplicity many ML concepts have been matched up with Java ones (for example, identifying packages with structures and subpackages with substructures) but only where it makes sense semantically (for example, static methods are like ML functions but non-static methods are not).

A secondary aim has been to facilitate the translation of Java code into ML. However, there are certain aspects of Java that cannot be simulated easily in ML (for example, the `protected` modifier). These limitations may be removed in a future revision but for the moment the priority is to make the interface simple.

7.2 Types

7.2.1 ML base types

Many ML base types are defined to be equivalent to Java types, as shown in Table 1. This permits passing values to and from Java without the need for coercions.

7.2.2 Java primitive types

A *primitive* Java type is one of the following: `int`, `bool`, `char`, `real`, `Int8.int`, `Int16.int`, `Int64.int` and `Real32.real`.

7.2.3 Java class types

There are two kinds of class types:

- *external* class types including those corresponding to ML base types such as `string` and `exn`;
- *internal* class types introduced by the `_classtype` construct described in Section 7.5.

External Java class types can be referred to using the same syntax as in the Java language, so for example `java.lang.StringBuffer` is a Java string buffer [3, §1.17] and `java.awt.Color` is a Java colour [4, §1.9]. This is possible because of the interpretation of Java packages as structures, and subpackages as substructures, as explained in Section 7.3.

7.2.4 Null values

Java class and array types, known collectively as Java *reference* types, are more refined than in the Java language, as they do *not* admit `null` as a value. This allows the compiler to assume, for example, that a field access operation cannot raise a `NullPointerException`. If *ty* is such a reference type, then values of type *ty* `option` are instances of the class or array (expressed as `SOME` *exp* for *exp* of type *ty*) and the Java value `null` (expressed as `NONE`).

7.2.5 Exporting and importing Java types

Whilst the non-`option` Java class or array types are useful within ML code that manipulates Java values, when a Java value reaches real Java code the information about presence or absence of `null` values is lost.

Suppose that we want to export an ML value to the Java world. This is possible only if the value has an *exportable type*, defined to be:

- A primitive Java type; *or*
- A Java class type; *or*
- A type *ty* `array` where *ty* is an exportable type; *or*
- A type *ty* `option` where *ty* is a Java class type; *or*
- A type *ty* `array option` where *ty* is an exportable type.

When a value is imported from the Java world stronger conditions are imposed because for class and array types the `null` value can always be passed into ML. Therefore an *importable type* is defined to be:

Java notion	*ML notion*
package	structure
subpackage	substructure
class name	type identifier
importing a package	opening a structure
static final field	value binding
static non-final field	value binding with `ref` type
static method	function binding
`void` type	`unit` type
multiple arguments	single tuple argument
exception class	exception

Table 2: Analogies between ML and Java

- A primitive Java type; *or*
- A type *ty* `option` where *ty* is a Java class type; *or*
- A type *ty* `array option` where *ty* is an importable type.

7.3 Static Java

To simplify access to Java classes, static fields and static methods, we make the analogies shown in Table 2. We consider each in turn.

7.3.1 Packages, subpackages, and classes

Top-level packages in the Java world are seen as a collection of top-level structures, and subpackages are substructures. Then classes are simply type identifiers inside structures. For example, the Java type `java.lang.Integer` is seen as an ML type identifier `Integer` inside a substructure `lang` inside a top-level structure `java`.

7.3.2 Import as open

The analogue to the Java `import` *package*`.*` construct is ML's `open` declaration; for example, `open java.lang` is roughly equivalent to `java.lang.*`. We also allow *classes* to be opened, giving unqualified access to static fields and methods (see later). Also, subpackages become visible as structures: after opening `java` it is possible to use `lang.Integer` to refer to the `java.lang.Integer` class.

7.3.3 Static fields

Static fields have little to do with object-oriented programming, so they are viewed as ML value bindings. Final fields really are treated as simple values, for example:

```
local open java.awt
in
  (* Note: this will be a Color option because it could be null... *)
  val redopt = Color.red

  (* ...though in fact, we're pretty sure that it's not! *)
  val red = valOf redopt
end

val neginf = java.lang.Double.NEGATIVE_INFINITY
```

Unfortunately, mutability for non-final fields in Java does not precisely correspond to ML's `ref` type; instead, it is closer to C's notion of *lvalue*. Still, it is possible to treat non-final fields as first-class refs by wrapping them inside objects that have appropriate read, write and equality test methods. At present, this isn't implemented properly and it is safest to dereference or assign to a non-final field immediately. Suppose some class `MyClass` has an integer field `x`:

```
fun increment () =
let val oldx = ! MyClass.x
in
  MyClass.x := oldx + 1
end
```

7.3.4 Static methods

As with fields, static methods have little to do with object-oriented programming, so they are viewed as ML value bindings of function type.

Void methods are interpreted as having `unit` result type; methods of no arguments have `unit` argument type. Methods with multiple arguments have a tuple argument type. There's one glitch: *overloading*. At present, method invocation must be written as an explicit function application, with an explicit tuple used for multiple arguments and an explicit `unit` value for methods with no arguments. MLj will attempt to resolve any overloading ambiguities and reject a program that it considers ambiguous. Because of the rather adhoc nature of its "attempt to resolve", seemingly unambiguous programs will sometimes be rejected. In this situation, insert a type constraint as close as possible to the function application.

Like Java, MLj will automatically apply upcasts (from a class to a superclass or superinterface) to arguments, and uses the same rules to pick the 'most specific' method where it is overloaded. In addition, it will automatically coerce arguments from type *ty* `option` to type *ty* where *ty* is a Java reference type. The example `getClass` below illustrates this.

```
(* Implement cosine using Java's own function *)
(* in java.lang.Math *)
local open java.lang
in
  (* Type constraint necessary because abs is overloaded *)
  fun abs (x : real) = Math.abs x
```

```
    (* Notice use of unit type *)
    val t = System.currentTimeMillis ()

    (* Notice how argument is coerced from string to string option *)
    (* Result must be coerced explicitly *)
    fun getClass (s : string) =
      case Class.forName (s) of
        NONE => raise Fail ("No such class: " ^ s)
      | SOME c => c
  end
```

A future version of MLj will permit static methods to be used in a first-class way, provided that overloading has been resolved.

7.3.5 Exceptions

Java exception classes can be bound to Standard ML exception identifiers using ML's existing exception declaration construct. For example:

```
exception IllegalArgExn = java.lang.IllegalArgumentException

(* Catch any exception that subclasses IllegalArgExn *)
fun test x = (do_some_java x)
  handle (e as IllegalArgExn) => do_something e

(* This exception cannot carry a string *)
fun fail () = raise IllegalArgExn
```

To be more precise, if the right hand side of the exception declaration is not a valid ML exception, then it is interpreted as a type expression denoting a class type that subclasses `java.lang.Exception` (which itself is equivalent to `exn`).

7.3.6 Exporting structures

The previous sections have explained how the static members of external Java classes can be accessed from ML. What about the converse: creating new Java classes inside ML? For static members, this is very straightforward. Any top-level Standard ML structure can be *exported* as a Java class, by interpreting its signature in the following way:

- Value bindings with function type are exported as static methods with the same name, provided that the function's argument type is `unit` (interpreted as `void`), a single importable type, or a tuple of importable types (interpreted as multiple arguments), and the result is either `unit` or a single exportable type.
- Value bindings with exportable types are exported as static final fields with the same name.

7.4 Object-oriented Java

We now get to the real object-oriented meat of the Java language.

7.4.1 Method invocation and field access

To deal with non-static method invocation and non-static field access, we introduce some new syntax:

exp.#*id*

Here *exp* is an expression (usually parenthesised unless a simple identifier) and `id` is a Java field or method name. The type of the expression *exp* must be a Java class; if not, MLj will reject the whole expression as badly typed.

The types of fields and methods are interpreted as for static methods. For example:

```
(* Static method invocation: string to Integer *)
val myInt = java.lang.Integer.getInteger("37")

(* Non-static method invocation: Integer to string (option) *)
val stropt = myInt.#toString ()

(* Non-static non-final field access; parentheses necessary *)
val x = ! (myObj.#x)

(* Non-static final field access *)
val y = myObj.#y
```

MLj uses the same rules as Java to resolve overloading and to apply coercions, with the addition of the `option` coercions already mentioned. In particular, objects inherit fields and methods from their superclass and superinterfaces.

7.4.2 Object creation

Syntax (if $n = 1$ the parentheses can be omitted):

`_new` *ty* ($exp_1, \ldots, exp_n$)

The `_new` construct corresponds to Java's class instance creation expressions [2, §15.8]. The type *tyarg* must resolve to a class type. This class is searched for accessible constructor methods using the same matching rules as for method invocations. The result of the whole expression has type *tyarg* so cannot be null. Here is an example:

```
fun generator i =
  _new java.util.Random.Generator (Int64.fromInt i)
```

7.4.3 Casting and class membership

A new syntax is introduced (borrowed from O'Caml) to denote casting up or down between Java classes:

exp :> *ty*

Examples:

```
(* Treat a string as an object; type constraint is necessary *)
fun stringToObject (x : string) = x :> java.lang.Object

(* We know this is object is actually a string *)
(* If it's not, then ClassCastException gets raised *)
fun objectToString (x : java.lang.Object) = x :> string
```

To test an object for membership of a class, we introduce a `_instanceof` construct analogous to Java's construct with the same name:

`_instanceof` *ty exp*

The type *ty* must resolve to a Java reference type, and the expression *exp* must have a Java reference type. The whole expression has type `bool`. Examples:

```
fun test (x : java.lang.Object) =
  if _instanceof java.lang.String x
  then "it's a string"
  else "it's not a string"
```

7.4.4 Synchronization

A construct for synchronization on an object is provided, analogous to that found in Java:

`( _synchronized` *exp* `)` *exp*

The first expression must have a Java class type or an ML `ref` type. Example:

```
(* Atomic increment operation *)
fun inc (x : int ref) =
  (_synchronize x)
  (x := !x + 1)
```

7.5 Class and interface declarations

The syntax of declarations is extended in two ways: [3]

- The new `_classtype` construct permits the definition of Java classes whose methods are written in ML and whose methods and fields may have ML types.
- The new `_interfacetype` construct permits the definition of Java interfaces from inside ML. This is really just a convenient way of avoiding the Java language (and compiler) if interfaces are required.

Both forms of declaration may only appear at the level of *structure* declarations, that is, whereever a `structure` declaration can appear.

These declarations can also appear as specifications in SML signatures, with additional syntactic and semantic restrictions as explained below. The syntax of class declarations is as follows:

$$\begin{array}{l}\langle \textit{classmods}\rangle\ \texttt{\_classtype}\ \textit{tycon}\ \langle\texttt{\_extends}\ \textit{ty}\rangle\\ \quad\langle\texttt{\_implements}\ \textit{ty}_1\,,\ldots,\textit{ty}_n\rangle\\ \quad\texttt{\{}\ \textit{classitem}_1\ \ldots\ \textit{classitem}_m\ \texttt{\}}\end{array}$$

The `_classtype` construct introduces a new class type *tycon* whose fields and methods are defined using ML types and expressions. It has similar functionality to the Java language `class` construct but can only be used to define non-static fields and methods; if you want to export static fields and methods then use the "structure-as-class" idea described in the previous section.

The superclass of the new class is specified by the class type *ty* (if not present then `java.lang.Object` is assumed), its superinterfaces by the interface types $ty_1, \ldots, ty_n$, and its fields, methods and constructors by the declarations appearing between braces.

The optional *classmods* is a sequence of distinct modifiers chosen from `_abstract`, `_final` and `_public`, whose meaning is the same as in the Java language.

7.5.1 Field declarations

Syntax:

$$\textit{classitem} ::= \langle \textit{fldmods}\rangle\ \texttt{\_field}\ \textit{id}\ \texttt{:}\ \textit{ty}$$

This declares a non-static field *id* with type *ty*.

The optional *fldmods* is a sequence of distinct modifiers chosen from `_final`, `_transient`, `_volatile`, `_public`, `_private` and `_protected`, whose meaning is the same as in the Java language.

Examples:

[3] This aspect of the extensions to ML syntax is subject to change in future versions of MLj; probably it will have something of the flavour of O'Caml

```
_classtype C
{
  (* A pair of integers stored in a Java instance variable *)
  _private _field pair : int*int

  (* The assignment to pair is obligatory *)
  _private _constructor (x : int, y : int)
    { _super(); pair = (x,y) }
}
```

7.5.2 Method declarations

Syntax:

$$\textit{classitem} \quad ::= \quad \langle \textit{methmods} \rangle \ \texttt{\_method}\ \textit{id}\ \texttt{(}\ \langle \textit{id}_1\ \texttt{:} \rangle\ \textit{ty}_1\,\texttt{,}\ \ldots\ \texttt{,} \langle \textit{id}_n\ \texttt{:} \rangle\ \textit{ty}_n\ \texttt{)} \\ \langle \texttt{:}\ \textit{ty} \rangle\ \langle \texttt{=}\ \textit{exp} \rangle$$

This declares a method *id* whose arguments are $id_1, \ldots, id_n$ with types $ty_1, \ldots, ty_n$, optional result type *ty* and optional body *exp*. A result type of `unit` is converted into `void`, the Java equivalent; alternatively, if *ty* is omitted then `void` is assumed.

Explicit argument types are required in order to type check simultaneous method declarations in the presence of Java's overloading and overriding.

The optional *methmods* is a sequence of distinct modifiers chosen from `_abstract`, `_final`, `_public`, `_private`, `_protected` and `_synchronized`, whose meaning is the same as in the Java language.

An abstract method must not have a body *exp*; for a non-abstract method, the body must be present.

Inside the method the identifier `this` refers to the object with which it was invoked and has the type of the defining class.

Example:

```
_classtype C
{
  _private _field pair : int*int

  (* Override the instance method from Object; notice return type *)
  _public _method toString () : string option =
  let val (x,y) = ! (this.#pair)
  in
    SOME ("(" ^ Int.toString x ^ "," ^ Int.toString y ^ ")")
  end
}
```

7.5.3 Constructor declarations

Syntax:

$$\textit{classitem} \quad ::= \quad \langle \textit{conmods} \rangle \ \texttt{\_constructor}\ \texttt{(}\ \langle \textit{id}_1\ \texttt{:} \rangle\ \textit{ty}_1\,\texttt{,}\ \ldots\ \texttt{,} \langle \textit{id}_n\ \texttt{:} \rangle\ \textit{ty}_n\ \texttt{)} \\ \texttt{\{}\ \textit{inits}\ \texttt{\}}\ \langle \texttt{=}\ \textit{exp} \rangle$$

where:

$$
\begin{array}{lcl}
\textit{inits} & ::= & \texttt{\_this}\ \textit{args} \\
 & | & \texttt{\_super}\ \textit{args}\ \langle\texttt{;}\ \textit{fldinits}\rangle \\
\\
\textit{fldinits} & ::= & \textit{id}\ \texttt{=}\ \textit{exp}\ \langle\texttt{;}\ \textit{fldinits}\rangle
\end{array}
$$

This declares a constructor for the class with optional modifiers *conmods*, arguments $id_1, \ldots, id_n$ with types $ty_1, \ldots, ty_n$, an initialiser block *inits* and an optional body *exp*.

The constructor first specifies whether it invokes another constructor in the same class (expressed as `_this` *args*) or a superclass constructor (expressed as `_super` *args*). This is similar to the Java language syntax for explicit constructor invocations [2, §8.6.5].

In addition, MLj requires that constructors initialise the fields. For fields with importable Java type this is not necessary: they will be initialised to the default value associated with that type [2, §4.5.4]. For all other fields, it is necessary to list their initial values explicitly in *fldinits*. Inside the field initialisers the keywords `_super` refers to the created object when considered to be an instance of its superclass.

Inside the body both `_super` and `_this` are available, as for methods.

Examples:

```
_classtype C
{
  _private _field pair : int*int
  _private _field obj : java.lang.Object option

  (* obj gets default initial value of null *)
  _private _constructor (x : int, y : int)
    { _super(); pair = (x,y) }

  (* Call existing constructor with default value for pair;
     set object field; print diagnostic message in body *)
  _public _constructor (obj : java.lang.Object)
    { _this(5,6); obj = SOME obj } = print "constructor called"
}
```

7.5.4 Superclass method invocation

Within a non-static method body one often wants to invoke a method from the superclass, bypassing any overriding of the method in the enclosing class definition. In Java, one uses the keyword `super` but this is given different semantics for field access and method invocation. In MLj we take a different approach and introduce a special syntax for superclass method invocation:

$$\textit{exp}\texttt{.\#\#}\textit{id}$$

Example:

```
_classtype D _extends C
{
  _field z : int

  (* Firstly invoke the superclass method *)
  _public _method toString () : string option =
  SOME (valOf (this.##toString ()) ^
    " and z = " ^ Int.toString (this.#z))
}
```

7.5.5 Exporting classes

From the compilation envoronment the command `export` specifies external names for classes defined using the `_classtype` construct. To safeguard against run-time errors such as `NullPointerException` or the faking of ML values in Java code, MLj restricts such classes in the following ways.

- For any field not declared private, it must:
  - *either* be declared final and have an exportable type;
  - *or* have an importable type.

  Rationale: non-private final fields cannot be updated but can be accessed from Java code, so therefore should contain only Java values. Non-final fields can also be updated, so should explicitly be allowed to take null values.

- For any method or constructor not declared private, its arguments must have importable types, *and* it must:
  - *either* be declared final and have an exportable return type;
  - *or* have an importable return type.

  Rationale: non-private methods can be invoked from Java code with arbitrary Java values, so the types of their arguments should be Java types that explicitly permit null values. Likewise the return type must be a Java type. Non-final methods can be overridden by Java-defined methods that sometimes return null values so their type must reflect this; final methods cannot be overridden.

There are no restrictions on the fields and methods of classes not listed by `export`: because their names are chosen by the compiler, it is assumed that external Java code cannot access them.

7.5.6 Interfaces

Syntax:

$\langle$*classmods*$\rangle$ `_interfacetype` *tycon* $\langle$`_extends` $ty_1, \ldots, ty_n\rangle$
`{` $intitem_1 \; \ldots \; intitem_m$ `}`

The `_interfacetype` construct introduces a new interface type *tycon* whose methods are declared using ML types. It has similar functionality to the Java language `interface` construct except that fields cannot be declared.

The superinterfaces of the new interface are specified by the interface types $ty_1, \ldots, ty_n$, and its methods by the declarations appearing between braces.

The modifiers *classmods* are more restrictive than for class types: only `_abstract` and `_public` are permitted. In fact, `_abstract` is superfluous because interfaces are implicitly abstract.

Each interface item *intitem* is a method declaration with the syntax shown below:

$$\mathit{intitem} ::= \langle \mathit{methmods} \rangle \ \texttt{\_method} \ \mathit{id} \ \texttt{(} \ \langle \mathit{id}_1 \ \texttt{:} \rangle \ \mathit{ty}_1 \ \texttt{,} \ \ldots \ \texttt{,} \langle \mathit{id}_n \ \texttt{:} \rangle \ \mathit{ty}_n \ \texttt{)} \ \langle \texttt{:} \ \mathit{ty} \rangle$$

Interface method declarations are like abstract methods in class types, but more restrictive: the only modifiers that are permitted are `_abstract` (again superfluous) and `_public`.

7.5.7 Signatures and signature matching

Class type and interface type declarations can appear in signatures, with the following restrictions:

- Methods must not have bodies and constructors must have neither initialisers nor bodies.
- Private fields, methods and constructors must not appear (rationale: they are not accessible outside the class definition).
- Methods that override a method in the superclass or superinterfaces must not appear (rationale: the superclass and superinterfaces imply their existence somewhere in the inheritance hierarchy so nothing is gained by redeclaring them).

When matching a class (interface) type in a structure against a class (interface) type in a signature all parts of the declaration must match exactly, modulo the restrictions listed above.

8 The Standard ML Basis Library

The following sections correspond to those listed on the Basis Library web page [1]. A summary is shown in Table 3.

8.1 Top-level Environment

All types and values in the top-level environment are implemented as specified, except for `use`.

Structure	Omissions and discrepancies
top-level	no `use`
`Array`	none
`Bool`	none
`BoolVector`,`BoolArray`	none
`Char`=`WideChar`	`Char.maxOrd` = 65535
`CharVector`=`WideCharVector`,`CharArray`	none
`CommandLine`	
`Date`	none
`General`	none
`IEEEReal`	`setRoundingMode` only accepts `TO_NEAREST`
`Int8`,`Int16`, `Int`=`Int32`,`FixedInt`=`Int64`	`~`, `*`, `+`, `-`, `div`, `quot`, `abs` don't raise `Overflow`
`Int`$\{N\}$`Vector`,`Int`$\{N\}$ `Array`	none
`LargeInt`=`InfInf`	none
`IO`	none
`List`	none
`ListPair`	none
`Option`	none
`OS`	none
`OS.FileSys`	no `chDir`, `isLink`, `modTime`, `readLink`, `realPath`, `setTime`, `tmpName`, `access` with `A_EXEC`
`OS.Path`	none
`OS.Process`	`getEnv` accesses Java properties
`Real`=`Real64`=`LargeReal`	no `fromDecimal`, `toDecimal`
`RealVector`,`RealArray`	none
`String`=`WideString`	none
`Substring`=`WideSubstring`	none
`TextIO`	no `openAppend` or `StreamIO` and related functions
`Time`	none
`Timer`	only wall-clock time is measured
`Vector`	none
`Word8`,`Word`=`Word32`, `LargeWord`=`Word64`	none
`Word`$\{N\}$`Vector`,`Word`$\{N\}$ `Array`	none

Table 3: Basis structures implemented in MLj

8.2 General

The structures `General`, `Option` and `Bool` are implemented in full. The structure `SML90` is not implemented.

8.3 Text

The structures `Char`, `String` and `Substring` are implemented in full. Contradicting the Basis definition, the value of `Char.maxOrd` is not 255, but 65535. Also, the structure `Char` is identical to `WideChar`, structure `String` is identical to `WideString` and structure `Substring` is identical to `WideSubstring`.

8.4 Integer

Structures `Int8`, `Int16`, `Int32` and `Int64` implement 8-bit, 16-bit, 32-bit and 64-bit signed integers respectively. The structure `Int` is synonymous with `Int32`, and `FixedInt` with `Int64`.

In all structures with signature `INTEGER`, the functions `~`, `*`, `div`, `quot`, `+`, `-` and `abs` do not raise `Overflow` when the result is not representable, instead 'wrapping round' in an unspecified manner. The functions `fromLarge`, `toInt`, `fromInt`, `fromString` and `scan` do raise `Overflow` when appropriate.

The structure `IntInf` (synonym `LargeInt`) implements infinite precision integers.

The structures `Word8`, `Word32` and `Word64` implement 8-bit, 32-bit and 64-bit unsigned integers respectively. The structure `Word` is synonymous with `Word32`, and `LargeWord` with `Word64`.

Structures with signature `PACK_WORD` are not yet implemented.

8.5 Reals

The structure `Real64` (synonyms `Real` and `LargeReal`) is almost completely implemented, with the exception of the functions `fromDecimal` and `toDecimal`.

The structure `IEEEReal` is implemented except for `setRoundingMode`, which raises a `NotImplemented` exception for modes other than `TO_NEAREST`.

Structures with signature `PACK_REAL` are not yet implemented.

8.6 Lists

Structures `List` and `ListPair` are implemented in full.

8.7 Arrays and Vectors

Structures `Array` and `Vector` are implemented in full. All appropriate monomorphic vectors and arrays are implemented in full. The structure `CharVector` is identical to `WideCharVector`.

Two-dimensional arrays will be introduced in a future release.

8.8 IO

The structure `IO` is implemented in full. The structure `TextIO` is mostly implemented with the exception of `TextIO.openAppend`, the substructure `StreamIO` and functions involving its types. Thus no low-level I/O is implemented. No other structure (or functor!) is yet implemented.

8.9 System

The top-level functions in structure `OS` are implemented.

The `OS.Process` structure is implemented in full, with the type `status` bound to `int` and values `success` and `failure` bound to `0` and `1` respectively. The function `OS.Process.system` can be used to execute arbitrary operating system commands, but at present there is no way of accessing the standard input, output and error streams associated with such a process. The `OS.Process.getEnv` function provides access to Java *properties*, as defined in the Java API [3, §1.18.8] and Language Specification [2, §20.18.7]. If you want to pass in Unix or Windows environment variables as properties then use the `-D` option under the `java` interpreter.

The `OS.FileSys` structure is partially implemented, with the following omissions: `chDir`, `isLink`, `readLink`, `realPath`, `modTime`, `setTime` and `tmpName`. Calling `access` with `A_EXEC` will raise an exception.

The `OS.Path` structure is implemented in full.

No other System structures are implemented.

8.10 Posix

The `Posix` structure is not implemented; Java is available on non-Posix platforms so this structure is not relevant.

9 Known bugs and omissions

- No warning is given for redundant matches, and incorrect warnings are sometimes issued for nonexhaustive patterns that are actually exhaustive.
- The restrictions on datatypes and exceptions as described in Section 6.3 are not enforced properly.
- The composition of conversion functions is not implemented efficiently as suggested by the basis (*e.g.* `Int64.fromLarge o Int32.toLarge`).
- The conversion functions `Word`X`.fromString` and `Word`X`.scan` do not raise `Overflow`.

References

[1] E. R. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library reference manual.* In preparation, but currently at `http://www.cs.bell-labs.com/~jhr/sml/basis/`.

[2] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[3] J. Gosling and F. Yellin. *The Java Application Programming Interface, Volume 1: Core Packages.* Addison-Wesley, 1996.

[4] J. Gosling and F. Yellin. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets.* Addison-Wesley, 1996.

[5] R. Harper and C. Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU-CS-96-136R, School of Computer Science, Cargnegie Mellon University, September 1996.

[6] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Mass., 1997.

[7] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1996.