CMB
The SML/NJ Bootstrap Compiler

(for SML/NJ version 110.35 and later)

User Manual

Matthias Blume
Lucent Technologies, Bell Labs

July 19, 2001

Contents

1 Introduction

With the exception of the runtime system which is written in C, compiler and interactive system for Standard ML of New Jersey (SML/NJ) [AM91] are themselves implemented in Standard ML [MTHM97]. When the new Compilation and Library Manager (CM) [Blu00] was introduced, all ML code of SML/NJ was reorganized in such a way that the system itself works like any other stand-alone ML program compiled by SML/NJ.

However, there are a few important differences between compiling an ordinary ML program using CM and (re-)compiling the interactive system itself. These differences are handled by a special-purpose version of the compilation manager: the bootstrap compiler. Its interface is `structure CMB`, exported from library `$smlnj/cmb.cm`.

This document describes how to use the bootstrap compiler and also explains how bootstrapping the SML/NJ compiler differs from compiling ordinary ML code.

2 Basic usage

2.1 Requirements

To be able to use the bootstrap compiler, one must first install SML/NJ (i.e., the interactive system that contains the compiler) as well as both ML-Yacc and ML-Lex.

It is further necessary to have all ML source code for the system available. If the basic installation did not install this source code, then one must now fetch all source archives and unpack them. (There is an option for the SML/NJ installer that lets it install all source trees automatically. However, by default this feature is turned off.)

The following list shows all required source packages. (Path names are shown relative to the SML/NJ installation directory.)

src/system This archive contains the sources for the SML Standard Basis library as well as lots of “glue” code. The glue is used for assembling a complete system from all its other parts. Directory `src/system` must be the current working directory at the time the bootstrap compiler is started.

src/MLRISC This is the implementation of the MLRISC backend (the low-level optimizer and code generator) used by SML/NJ.

src/cm This source tree contains most of CM’s (and CMB’s) implementation.

src/compiler This is the implementation of the frontend (parser, type checker, etc.) of the compiler as well as its high-level optimizer (FLINT).

src/ml-yacc This source tree contains the implementation of ML-Yacc and its library (`$/ml-yacc-lib.cm`). Technically, the sources of ML-Yacc itself are not needed (provided a working executable for ML-Yacc has been installed), but the library sources are.

src/smlnj-lib This source tree hosts several sub-trees, each of which implements one of the libraries in the SML/NJ library collection. For bootstrap compilation, the following sub-trees are required:

- **src/smlnj-lib/Util** This directory holds the sources for `$/smlnj-lib.cm`.
- **src/smlnj-lib/PP** This directory holds the sourcse for `$/pp-lib.cm`, i.e., the pretty-printing library.
- **src/smlnj-lib/HTML** This directory holds the sources for `$/html-lib.cm`, a library for handling HTML files. The need for this library arises from the fact that `$/pp-lib.cm` statically depends on it. (The compiler does not actually use any services from this library.)

2.2 Invoking the bootstrap compiler

To invoke the bootstrap compiler, first one has to change the current working directory to `src/system`:

```
$ cd src/system
```

The next step is to start the interactive system and load the bootstrap compiler. This can be done in one of two ways:

1. Start the interactive system and then issue a `CM.autoload` command that causes the bootstrap compiler to be loaded. The resulting session could look like this:

   ```
   $ sml
   Standard ML of New Jersey ...
   - CM.autoload "$smlnj/cmb.cm";
   ...
   val it = true : bool
   -
   ```

2. Start the interactive system and specify `$smlnj/cmb.cm` on the command line:

   ```
   $ sml '$smlnj/cmb.cm'
   Standard ML of New Jersey ...
   -
   ```

Note for frequent compiler hackers: The `makeml` script (see below) builds the the interactive system in such a way that `$smlnj/cmb.cm` is already pre-registered for autoloading. Therefore, when using an interactive system built by `makeml` (as opposed to the original config/install.sh) there is no need for loading the bootstrap compiler explicitly.

At this point one can invoke the bootstrap compiler by simply issuing the command `CMB.make()`:

```
- CMB.make ();
```

If `CMB.make()` does not run to successful completion, you do not have to start from the beginning. Instead, fix the problem at hand and re-issue `CMB.make()` without terminating the interactive session in between. This tends to be a lot faster than starting over.

This process can be repeated arbitrarily many times until `CMB.make()` is successful.

A successful run looks like this:

```
- CMB.make ();
...
New boot directory has been built.
val it = true : bool
-
```

The return value of `true` indicates success. This means that (as indicated by the message above the return value) a directory with stable libraries and some other special files that are needed for “booting” a new interactive system has been created.

The name of the boot directory depends on circumstances. A part of it can be chosen freely, other parts depend on current architecture and operating system. For example, on a Sparc system running some version of Unix, the default name of the directory is `sml.boot.sparc-unix`.

There is also a similarly-named em binfile directory which is used by `CMB.make()` itself but which is not required for the purpose of subsequent “boot” steps.

2.3 Booting a new interactive system

Once `CMB.make()` has completed its work successfully (indicated by its return value of `true`), the next step is to build a new heap image. To do so, issue the command `makeml` at the shell prompt:

```
$ ./makeml
...
Standard ML of New Jersey ...
./makeml: Heap image generated.
```

The `makeml` command generates a new heap image and also prepares stable libraries to be used by this image. Neither the heap image nor the libraries will at this time be installed for permanent use though. This means that invoking `sml` still starts the old system.

Again, the name of the generated heap image is variable and depends on programmer choice, current architecture, and current operating system. For example, on an Intel x86 Linux machine, the default would be `sml.x86-linux`.[1]

2.4 Testing the newly booted system

To test-drive a newly booted system without installing it, issue the `testml` command at the shell prompt:

```
$ ./testml
```

This starts an interactive system in a way very similar to `sml`, but it uses the heap image and libraries from a previous run of `makeml`.

2.5 Installing a newly booted system

Once one is sure that a newly booted system is good enough to replace the old system, issuing the `installml` command will replace old heap image and old libraries with those generated by the previous run of `makeml`.

```
$ ./installml
```

This command will replace the system's heap image in `../../bin/.heap` and its libraries in `../../lib`. However, it will leave alone any unrelated libraries in `../../lib`.

Sometimes changes to the compiler will render any previously installed libraries unusable. In this case one should erase them prior to issuing the `installml` command:

```
$ rm ../../lib/*
$ ./installml
```

Libraries that were installed as part of the SML/NJ installation process but which are unrelated to bootstrap compilation (e.g., `$/inet-lib.cm`, CML, eXene) can be recovered (once they had been removed) by going back to the installation directory and issuing the `config/install.sh` command again:

```
$ cd ../..
$ config/install.sh
```

Since some changes to the compiler also render old binfiles unusable, one will occasionally have to remove those first (prior to re-running `config/install.sh`). Binfiles for libraries unrelated to bootstrapping are handled by CM (and not CMB), so the usual CM rules for locating them apply. (This means that in such a case the binfile for *d*/*f*`.sml` will be in *d*/`CM`/*arch*–*os*/*f*`.sml` where *arch* is a string describing the CPU architecture and *os* is a string describing the operating system kind. Example: `x86-unix`.)

[1] The operating-system dependent portion of the name of a heap image is more discriminating than the operating-system dependent portion of boot- or binfile directories. On the same Intel x86 Linux machine, the name of the boot directory is `sml.boot.x86-unix`.

3 Differences between CMB and CM

In this section we discuss why compiling the compiler is different from compiling other ML programs. Each of the following sub-sections focuses on one particular aspect.

3.1 Code sharing

CM keeps compiled code within the same directory tree that contains the corresponding ML source code. Thus, there is a fixed function that maps the names of source files to the names of corresponding binfiles and the names of CM description files (for libraries) to their corresponding stable files.

As a result, these files will be shared between programs that use the same libraries. Moreover, CM will let different programs that are loaded into an interactive session at the same time share their in-memory copies of common compiled modules. (There is also an issue of state-sharing, but this does not concern CMB because the bootstrap compiler only compiles code without linking it.)

Sharing of code is useful for ordinary usage, but when compiling the compiler itself, it is not desirable. During bootstrap compilation, it is often the case that several different versions of compiled code have to coexist. Some, or even all of these versions can differ significantly from those of the currently running system.

Therefore, CMB keeps binfiles and stable files in separate directory trees. The names of the directories where these trees are rooted at are constructed from three parts; the binfile directory's name is u`.bin.`*arch*–*os* and the stablefile directory's name is u`.boot.`*arch*–*os*. As mentioned before, *arch* is a string describing the current CPU architecture and *os* is a string describing the current operating system kind. Component u is a string that can be selected freely when the bootstrap compiler is invoked. When using `CMB.make` it defaults to `sml`, otherwise it is the argument given to `CMB.make'`. (The u component is kept variable to make it possible to keep and use several compiled versions of the system at the same time.)

The auxiliary script `makeml` (which is responsible for bootstrapping a new system) also accepts a parameters to select u. If the parameter is missing, it defaults to `sml` (in accordance with `CMB.make`'s behavior).

3.2 Init library

The *init library* (`$smlnj/internal/init.cmi`) is a library that is used implicitly by all programs. This library is "special" in several ways and cannot be described using an ordinary CM description file. It is the bootstrap compiler's responsibility to properly prepare a stable version.

Ordinary programs (those managed by CM) do not have to worry about the special aspects of how to construct this library; they just have to be able to use its stable version.

There are several reasons why the library cannot be described as an ordinary CM library:

- The library exports the *pervasive environment* which normally is imported implicitly by every compilation unit. Within the init library, no pervasive environment is available yet.
- One binding in the above-mentioned pervasive environment is a binding for `structure _Core`. The symbol `_Core` is not a legal SML identifier, and the bootstrap compiler has to take special action to create a binding for it anyway.
- One of the compilation units in this library is merely a placeholder which at link time has to be replaced by the SML/NJ runtime system (which is written in C).

3.2.1 Linkage to runtime system

The ML source file `dummy.sml` (located in directory `src/system/smlnj/init`) contains a carefully constructed module whose signature matches that of the runtime system's binary API. This file is being compiled as part of constructing the init library, but it has been marked specially as *runtime system placeholder*. During compilation, this file pretends to *be* the runtime system; other modules that use the runtime system "think" they are using `dummy.sml`.

At link time (i.e., bootstrap time—when `makeml` is run), the boot loader will ignore `dummy.sml` and use the actual runtime system in its place. This trick makes it possible that (from the point of view of all other modules) using services from the runtime system appears to be no different than using services from an ordinary ML compilation unit.

3.3 BOOTLIST file

Linking of SML/NJ programs involves executing the code of each of the concerned compilation units. The code of each compilation unit is technically a closed function; all its imports have been turned into arguments and all exports have been turned into return values.

For ordinary programs, this process is under control of CM; CM will take care of properly passing the exports of one compilation unit to the imports of the next.

When booting a stand-alone program, though, there is no CM available yet. Thus, executing module-level code and passing exports to imports has to be done by the (bare) runtime system. The runtime system understands enough about the layout of binfiles and library files so that it can do that—provided there is a special *bootlist* file that contains instructions about which modules to load in what order.

The bootlist mechanism is not restricted to building SML/NJ. Ordinary ML code can also be turned into stand-alone programs, and as far as the runtime system is concerned, the mechanisms are the same. The bootlist file used by such ordinary stand-alone ML programs will be constructed by CM; only in the case of bootstrapping SML/NJ itself it will be constructed by CMB.

The name of the bootlist file is `BOOTLIST`, and it is located at the root of the directory tree that contains stable files (i.e., its name is u`.boot.`*arch–os*`/BOOTLIST`).

3.4 PIDMAP file

The last file to be loaded by the bootstrap process contains module-level code which will trigger the self-initialization of the interactive system—including CM. One job of CM is to manage sharing of link-time state (i.e., dynamic state created by module-level code at link time). Link-time state of a module used by the interactive system should be shared with any program using the same module. The file u`.boot.`*arch–os*`/PIDMAP` contains information that enables CM to relate existing link-time state to particular library modules and also to identify any link-time state that will never be shared and which can therefore be dropped. It is CMB's responsibility to construct the `PIDMAP` file.

3.5 Cross-compiling

Several different versions of the bootstrap compiler can coexist—each being responsible for targeting another CPU-OS combination. Structure `CMB` is the default bootstrap compiler that targets the current system; it is exported from `$smlnj/cmb.cm`. The following table lists the names of other structures—those corresponding to various cross-compilers. All these structures share the same signature.

The table also shows the names of libraries that the structures are exported from as well as those *arch* and *os* strings that are used to name binfile- and stablefile-directory.

library	structure	architecture	OS	*arch*	*os*
`$smlnj/cmb.cm` `$smlnj/cmb/current.cm`	`CMB`	current	current		
`$smlnj/cmb/alpha32-unix.cm`	`Alpha32UnixCMB`	Alpha	Unix	`alpha32`	`unix`
`$smlnj/cmb/hppa-unix.cm`	`HPPAUnixCMB`	HP-PA	Unix	`hppa`	`unix`
`$smlnj/cmb/ppc-macos.cm`	`PPCMacOSCMB`	Power-PC	Mac-OS	`ppc`	`macos`
`$smlnj/cmb/ppc-unix.cm`	`PPCUnixCMB`	Power-PC	Unix	`ppc`	`unix`
`$smlnj/cmb/sparc-unix.cm`	`SparcUnixCMB`	Sparc	Unix	`sparc`	`unix`
`$smlnj/cmb/x86-unix.cm`	`X86UnixCMB`	Intel x86	Unix	`x86`	`unix`
`$smlnj/cmb/x86-win32.cm`	`X86Win32CMB`	Intel x86	Win32	`x86`	`win32`
`$smlnj/cmb/all.cm`	all of the above (except `CMB`)				

As an example, consider targeting a Sparc/Unix system. The first step is to load the library that exports the corresponding cross-compiler:

```
CM.autoload "$smlnj/cmb/sparc-unix.cm";
```

Once this is done, run the equivalent of `CMB.make`:

```
SparcUnixCMB.make ();
```

This will recompile the compiler, producing object code for a Sparc. Binfiles will be stored under `sml.bin.sparc-unix` and stable libraries under `sml.boot.sparc-unix`.

4 Structure CMB

This section describes the signature of `structure CMB`. Since structures representing cross-compilers have the same signature, everything said here applies (*mutatis mutandis*) to them as well.

The primary function to invoke the bootstrap compiler is `CMB.make'`:

```
val make' : string option -> bool
```

This (re-)compiles the interactive system's entire source tree, constructing stable versions for all libraries involved. In the process, binfiles are placed under directory u`.bin.`*arch*–*os* where *arch* and *os* are strings describing target architecture and target OS, respectively. The string u is the optional argument to `CMB.make'`. If set to `NONE`, it defaults to `"sml"`.

An alternative equivalent to invoking `CMB.make'` with `NONE` is to use `CMB.make`:

```
val make : unit -> bool
```

CMB—like CM—maintains a lot of internal state to speed up repeated invocations. (Between sessions, much of this state is preserved in those binfile- and stablefile-directories. However, reloading is still quite a bit more expensive than directly using existing in-core information.)

Information that CMB keeps in memory can be completely erased by issuing the `CMB.reset` command:

```
val reset : unit -> unit
```

After a `CMB.reset()`, the next `CMB.make` (or `CMB.make'`) will have to re-load everything from the file system.

CMB has its own registry of "CM identifiers"—named values that can be queried by using the conditional compilation facility. This registry is initialized according to CM's rules. Of course, initial values are not based on current architecture and OS but on those of the target system. To explicitly set or erase the values of specific variables, one can use `CMB.symval` (which acts in a way analogous to `CM.symval`):

```
val symval : string ->
             { get : unit -> int option, set : int option -> unit}
```

5 File naming—the role of path anchors

Under normal operation of CM, the mapping from path names to the files they denote is supposed to be a fixed one. The path anchor mechanism is merely a means of configuring this mapping according to the actual filesystem layout.

The bootstrap compiler and the associated boot procedure, however, use path anchors extensively for other purposes. In particular, the above-mentioned mapping will vary over time.

The "casual" compiler hacker does not actually need to remember all the details because these details are mostly taken care of automatically by `CMB` and its various scripts (`makeml`, `testml`, `installml`). But it is useful to have a rough idea of what rôles are being played by various files and directories that are being used or created.

5.1 Anchoring requirement

During bootstrap compilation it is necessary that all pathnames be either anchored or relative to another anchored pathname. This rule guarantees that every path name can be mapped to different locations in the filesystem at different times. (Technically, this restriction is necessary only for library description files.)

5.2 Different anchor mappings at different times

We can distinguish between four different anchor configurations which will be in effect at different times:

compilation: At the time `CMB.make` runs, the anchor configuration is taken from file `pathconfig`.[2] This configuration maps anchors to their respective directories in the actual source tree. At the same time the policy which determines the names of binfiles and stablefiles is modified in such a way that all binfiles will be created in directory u`.bin.`*arch–os* and all stablefiles will be created in u`.boot.`*arch–os*. In particular, if `$`a is the root anchor name controlling the path leading to an ML source file `$`a`/`$\cdots$`/`*file*`.sml`, then the corresponding binfile will be created as u`.bin.`*arch–os*`/`a`/`$\cdots$`/CM/`*arch–os*`/`*file*`.sml`. Similarly, the stablefile for the library description `$`a`/`$\cdots$`/`*file*`.cm` controlled by root anchor `$`a gets written to u`.boot.`*arch–os*`/`a`/`$\cdots$`/CM/`*arch–os*`/`*file*`.cm`. (Notice how the anchor name `$`a is being incorporated into the resulting pathnames.)

boot: At bootstrap time (e.g., when the `makeml` script is invoked), CM scans the contents of u`.boot.`*arch–os*, and for each directory name a it finds there it maps the anchor `$`a to that directory. The result is that the location for library description `$`a`/`$\cdots$`/`*file*`.cm` is considered to be u`.bin.`*arch–os*`/`a`/`$\cdots$`/`*file*`.cm`, which means that—under the *usual* rules of CM— the corresponding stablefile is going to be u`.bin.`*arch–os*`/`a`/`$\cdots$`/CM/`*arch–os*`/`*file*`.cm`. This is precisely where `CMB.make` created it. (Of course, the libary description file itself does not exist, but this is not a problem because the stablefile does.)

The resulting mapping for anchors is the one that is being saved to the newly generated heap image.

test: After generating a new heap image v`.`*arch–osname*[3], the `makeml` script will copy the hierarchy of directories under u`.boot.`*arch–os* to a new directory v`.lib` and generate a path configuration file v`.lib/pathconfig` for it. Files in this hierarchy will be re-created as hard links. The prefix v can be chosen at the time `makeml` is invoked (see section 6.1)—just like u can be chosen at the time `CMB.make` is invoked. The default for v is `sml`. Copying the directory prevents any future `CMB.make` from clobbering the libraries that belong to the newly generated heap image.

Running the `testml` script will start the runtime system, instruct it to load heap image v`.`*arch–osname*, and arrange for the path configuration file v`.lib/pathconfig` to take effect. This will cause any anchor a to be resolved within the v`.lib` hierarchy. Thus, the new system can be tested in a way that is completely independent from any existing installation. (Again, v can be specified as a parameter to `testml`.)

[2] All relative pathnames shown here are relative to `src/system`.

[3] As mentioned earlier, we distinguish between *os* which describes the *kind* of operating system and *osname* which is the name of a particular operating system. For example, on a Linux machine *osname* is `linux` while *os* is still `unix`.

install: When testing has been satisfactory, the new system can be installed permamently, replacing the old heap image and its old libraries with their newly created counterparts. To do so, one should run the `installml` script. It will move the heap image v`.`*arch–osname* to `../../bin/.heap/sml.`*arch–osname* and all stablefiles from v`.lib` to their usual place under `../../lib`. It then proceeds to edit `../../lib/pathconfig` (the main path configuration file of the installation) to reflect the new situation. (Library files in `../../lib` that have nothing to do with the bootstrap process will remain untouched.)

6 Scripts

This section gives a detailed description for each script, its function, and its options. All scripts described here are located in the `src/system` directory.[4]

6.1 The `makeml` script

This script is used after a successful run of `CMB.make` (or `CMB.make'`). It *links* the compiler and its interactive system, forming a corresponding heap image. In addition to that, it prepares a new directory containing stable CM libraries to be used with the new image.

One way of thinking about this is to view `makeml` as a function mapping a bootfile directory b to a pair consisting of a heap image v`.`*arch–osname* and a library directory v`.lib`. The strings b and v are optional parameters; the defaults are `sml.boot.`*arch–os* for b and `sml` for v.

The script accepts a number of options as follows:

-o v specifies a v other than the default `sml`.

-boot b specifies a b other than the default `sml.boot.`*arch–os*.

-quiet instructs `makeml` to greatly reduce its diagnostic output. In particular, the names of files being linked are not shown. The default for this can be controlled via a boolean-valued environment variable `MAKEML_VERBOSITY`. If the variable is not set, then the default is `true` (meaning "verbose").

-verbose is the opposite of `-quiet`.

-rebuild u puts `makeml` into a different mode: After loading the executable section of all binfiles and linking them, it will not read any static environments, will not initialize the usual interactive system and will not produce a heap image. Instead, it internally invokes the equivalent of `CMB.make' (SOME "`u`")`, thus recompiling everything again. When recompilation is complete, `makeml` stops; the newly-built system must be linked using a separate explicit invocation of `makeml`. Notice that u`.boot.`*arch–os* must be different from b.

-rebuildlight u is the same as `-rebuild` u except that the symbol `LIGHT` will be defined (using `CMB.symval`) for the duration of the compilation. The effect of this is that no cross-compilers will be built (which can save considerable time). Alternative names for **-rebuildlight** are **-light** and **-lightrebuild**.

-bare causes `makeml` to build a reduced version of the system without the compilation manager CM included. This is useful for people who are interested in an interactive system only.

-run r selects the executable for the SML/NJ runtime system. The default is ../../bin/.run/run.*arch-osname*.

-alloc a specifies the size of the SML/NJ garbage-collector's allocation area. The default depends on the current machine architecture.

The most common usage is to simply say `./makeml` without any arguments, taking advantage of the defaults as described above.

[4] If the current directory `.` is not on the shell's search path, then each script must be invoked using an explicit `./`-prefix. Example: `./makeml`.

6.2 The `testml` script

The `testml` script launches a newly-built and newly-linked interactive system. It requires a previous run of the `makeml` script because it uses the heap image v`.`*arch–osname* and libraries in v`.lib`.

If no arguments are given, v defaults to `sml`. If at least one argument is given, then v is assumed to be the first one. All other arguments are passed to the initialization routine of the interactive system just like normal arguments to the `sml` command would.

This means that one *must* specify a v unless there are no command-line arguments at all.

6.3 The `installml` script

The `installml` script moves a heap image v`.`*arch–osname* to `../../bin/.heap/sml.`*arch–osname*. Moreover, it moves the libraries under v`.lib` to `../../lib` and updates `../../lib/pathconfig` accordingly.

The script takes one optional argument which specifies v. If no argument is given, v defaults to `sml`.

6.4 The `fixpt` script

One good test of whether a new version of the compiler is working properly is to see if it compiles to a *fixed point*. For this, the compiler is compiled n times with the result of the $(k-1)$st compilation being responsible for compiling the kth version. A fixed point is reached if two consecutive compilations produce identical results.

The `fixpt` script automates the task of compiling to a fixed point. It internally uses the `sml` command for the first compilation and the `makeml` script with its `-rebuild` parameter for all subsequent runs. This produces a series of bin-directories u`.bin.`*arch–os*, u`1.bin.`*arch–os*, u`2.bin.`*arch–os*,... and boot-directories u`.boot.`*arch–os*, u`1.boot.`*arch–os*, u`2.boot.`*arch–os*,... where u is a common "stem" that is used for naming the whole series.

The `fixpt` script accepts the following options:

-iter n limits the number of iterations to n. The default for n is 3. If no fixed point is found after n iterations, then `fixpt` terminates with an error message.

-base u selects the "stem" u (see above). The default is `sml`.

-light causes `LIGHT` to be defined during compilation. (See the discussion of `makeml`'s `-rebuildlight` option.)

Failure to reach a fixed point after 2 iterations usually indicates some serious problem within the compiler.

6.5 The `allcross` script

Finally, the `allcross` script is handy for building bin- and boot-files for all architectures. The script currently takes no arguments and compiles (and cross-compiles) for 6 supported combinations of *arch* and *os*: `alpha32-unix`, `hppa-unix`, `ppc-unix`, `sparc-unix`, `x86-unix`, and `x86-win32`.

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[Blu00] Matthias Blume. CM: The SML/NJ compilation and library manager. Manual accompanying SML/NJ software, 2000.

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