Standard ML of New Jersey

—

Batch Compiler
(Version 0.93)

February 15, 1993

Standard ML of New Jersey
License and Disclaimer

1 Batch Compiler

Note: The batch compiler is obsolete after version 0.93 and will not be distributed with future versions of the compiler. This document is the section of the Release Notes for version 0.75 describing how to use the batch system.

The SML/NJ batch compiler provides some (unsafe) separate compilation and cross compilation capabilities. The batch system compiles files containing signature, structure, and functor declarations, i.e. only module level declarations are allowed. Compiling a module "Foo" produces a corresponding object file "Foo.mo", which contains the names of the other .mo files required to build the module, as well as the code of the module itself. Modules are compiled separately, producing the necessary .mo files.

A program is executed by passing a module name to the run-time system with the command (assuming src is current working directory):

`runtime/run` *[memory managment options]* `Module`

The run-time system starts a linker which loads in the module, finds out what other modules it depends on, recursively loads them and passes them to the original module. The module then "builds" itself, by executing its top-level declarations and creating a run-time structure. By convention, the loader looks for the module object files in the directory src/mo. In this case it would look for an object file named src/mo/Module.mo.

For example, batch compiling the following module will create a file named `HelloWorld.mo`.

```
structure HelloWorld =
  struct
    val foo = print "hello world"
  end
```

When `HelloWorld.mo` is placed in the directory `src/mo` and the command

```
% runtime/run HelloWorld
```

is executed, the run-time system will build a run-time record containing foo, and as a side effect of building the record, "hello world" will be printed.

Thus a program consists of the collection of all modules on which a given top-level structure (transitively) depends. The order in which the modules will be executed/created is determined by a postorder traversal of the (acyclic) dependence graph.

Using the batch system for separate compilation is unsafe since the .mo files contain no type information, only the names of the modules on which they directly depend. However, as long as each module has the correct signature, the run-time record will be type-correct; in other words, it is safe to edit, re-compile, and re-link modules as long their signatures do not change.

1.1 Installation

The compiler is composed of two parts: a run-time system written in C and assembly language; and a number of ML object files which form the main part of the compiler. The run-time system provides garbage collection, signal handling, and system calls; the rest of the compiler, including the standard library, is written in ML. The .mo files for the compiler reside in the mo.vax and mo.m68 subdirectories of the ML distribution. The source for the compiler resides in the src subdirectory, and these instructions assume that the current directory is src.

To build a batch compiler:

1. Go to the src subdirectory of mldist:

   ```
   % cd src
   ```

2. Run the makeml script with the -batch option, e.g.

   ```
   % makeml -batch
   ```

The other options for makeml have the same meaning as for the interactive system. See the man page doc/makeml.1.

As the command executes, you should see messages like "[Loading Foo]" and "[Executing Bar]", which indicate the modules being linked in, and then messages like "signature ASSEMBLY" and "functor CoreFunc", which indicate that the standard library is being compiled.

When the command successfully terminates, an executable image named "batch" has been created in the current directory. This is the batch ML compiler. If you wish to use a different name for this file, use the -i option of makeml.

Now you can run the batch compiler by typing "batch". You can give it commands interactively, or by redirecting its input from a file, as is done below with the command script "all".

Normally you will want to build a complete interactive or batch system, but there are occasions when you may want to build just the run-time system. The option -run of makeml will do this.

When invoked, `runtime/run` takes the name of a module (say "Foo") as a parameter. It looks for a file with path name mo/Foo.mo relative to the current directory and loads that file, and then recursively loads mo/Bar.mo for all modules Bar on which Foo depends. Finally the code for creating the module Foo is executed. For instance, to run the interactive system on a Vax, one could symbolically link mo to mldist/mo.vax (or ../mo.vax if current directory is src) and execute the command

```
runtime/run IntVax
```

This causes mo/IntVax.mo to be loaded, together with all the modules on which it depends (essentially the whole compiler). IntVax.mo (defined in src/build/glue.sml) causes the interactive top-level loop to be executed when the functor Interactive is applied. Any program can be directly loaded and executed by the run-time system in this manner, not just the compiler.

The run-time system accepts the option flags -h, -r, -m, and -g to control heap sizing and garbage collection messages.

1.2 Using the batch compiler

The batch compiler accepts commands on its standard input. The list of commands is:

`!file`	compile the file.
`*file`	assemble the file.
`<file`	parse the file.
`>file`	export to a file.
`%`	print the last generated lambda (intermediate code).
`#word`	comment; ignored.
`@directory`	look for files in a directory. directory should end in
`~function`	execute a function.
`^flag`	toggle a flag.
`?`	print this help message.

There should be no space between the command character and the argument string, which should not be quoted.

The execute "~" and toggle "^" commands are mainly used to control debugging facilities, which are not explained here. Typing "^" alone on a line produces a list of possible flags.

The compile command, "!", causes the named file to be compiled, generating an object file A.mo for each module A defined in the file. The assemble command, "*", generates an assembly listing A.s that can be assembled to produce the corresponding .mo file. The parse command, "<", causes the file to be parsed and type-checked, but produces no output files. These three commands all update the symbol table with the bindings defined in the file. The file must contain only signature, structure, and functor declarations.

The export command ">" exports the current state of the batch compiler to an executable file with the given name. This is a way of preserving the symbol table state of the compiler, and is useful for separate compilation; if you change a module without changing its signature, you can safely recompile it using the exported compiler (which contains the symbol table information from other modules that the modified module may require). For example, the file "all" in the src directory is a batch command script for compiling the compiler. The last command in the file exports to a file "upto.all", which can recompile any module of the compiler.

Once you have compiled all of your code, you only need to move it to the mo directory and execute it in the same way as CompM68 or CompVax:

```
runtime/run Foo
```

for a module Foo. Note that the files CoreFunc.mo, Math.mo (for Vax), Initial.mo, and Loader.mo must be in the mo directory, as they are used in bootstrapping the system.

2 Bootstrapping and Recompiling the Compiler

`runtime/run` is the ML loader. It searches for things it needs in the mo directory. It takes one argument, which is the name of a structure. That structure is found in the mo directory and executed for side effects.

`runtime/run` loads only 4 .mo files:

```
CoreFunc.mo  Math.mo Initial.mo Loader.mo
```

It passes its argument (a structure name) to Loader.mo, which then loads many other .mo files (the entire DAG of dependencies whose root is CompFoo.mo (or whatever)).

The mo directory contains mo files. These carry names of top-level structures or functors. They are copies of the output produced by the code generator. As such they are machine-dependent. There is no software support for figuring out to which code generator (or machine) a .mo file corresponds. Thus it is entirely up to the user to make sure that the .mo files in the .mo directory make sense and are the right ones for the task at hand.

The highest-level structure of a mo file is: fn () => (["a","b","c"], fn [a,b,c] => top-level structure or functor) where the ["a", "b", "c"] is a list of names of structures on which the top-level thing depends, and the [a,b,c] are the structures themselves. Thus a, b, and c are the only free variables in the top-level structure, and the thing in the mo file is actually a closed lambda-term. Only the run program reads .mo files.

The CompFoo structure (created with the Batch functor) is a structure that executes a small batch compiler as a side effect. IntFoo is a structure that executes the full interactive system as a side effect. The only difference between the two is in the user interface. The 'run' loader is typically used only on one of these two structures, to execute either the batch or the interactive system. Both the batch and interactive systems support an 'export ML' command that saves the currently executing system into a file in a.out format, so that it can be re-executed at will.

The batch system has three interesting commands that deal with ML source. They are:

```
!file   compile file, and write a .mo file for each top-level structure
*file   compile file, and write a .s file for each top-level structure
<file   read in and parse the file
```

All three commands enter appropriate things into the batch compiler's symbol table, so that later files can refer to them, and the appropriate top-level references can be resolved.

It is possible to set batch options with ^option The options ^printit and ^saveLvarNames will cause useful intermediate code to be written to the .s file, in addition to the assembly code.

Making a new compiler

Use the appropriate batch to generate appropriate .mo files. Then use the boot loader (run) to generate a new compiler. There it is.

Baby steps

When debugging the mips code generator, it might be worth replacing CoreFunc with a very simple structure ("hello world"), then using batchm to generate a fake CoreFunc.mo. runtime/run would execute CoreFunc before dying in Math or Initial.

NOTE: Recompiling the compiler will require a heap size of at least 6MB, so it if possible it should be done on a machine with at least 8MB of physical memory. Use the -m flag of runtime/run to set softmax as high as reasonable (e.g. -m 6000 on an 8MB Sun-3).

NOTE: Beware of using the sharable run-time system when recompiling the compiler, since it has the effect of embedding old versions of the .mo files into the system. Either use a nonsharable version of runtime/run, or rebuild a sharable runtime/run after each iteration of compiling the compiler to incorporate the latest .mo files.

If you change the compiler, you can use the batch command script "all" to recompile it with the batch system:

```
batch < all
```

or

```
runtime/run {CompM68 | CompVax | CompSparc | CompNS32} < all
```

This will generate new .mo files for all the modules in the compiler including the "boot" modules (whose source files are found in src/boot):

```
CoreFunc.mo  [boot/core.sml]
Math.mo     [boot/math.sml]
Initial.mo  [boot/perv.sml]
Loader.mo   [boot/loader.sml]
```

These are generated by the `~mBoot` command in the all script rather than using the `!` compile command. These files are treated differently because of their role in bootstraping the pervasive (i.e. built-in) environment.

The newly generated .mo files should be moved into a new mo directory, say mldist/mo.new:

```
cd ..   (to mldist)
mkdir mo.new
mv src/*.mo mo.new
```

After generating new .mo files for the compiler (including the boot modules if necessary), you should build a new batch compiler by symbolically linking mldist/mo.new to mldist/src/mo and recompiling the source code again (using the -mo option to makeml to designate the object directory).

```
% cd src
% makeml -batch -mo ../mo.new -vax bsd ...
% batch < all
```

The resulting .mo files should be identical to the previous set of .mo files in mo.new. Sometimes more than one iteration is necessary (for example if the signature of one of the boot modules has changed). The bootstrapping is successful when two successive iterations produce identical .mo files.

The last command in the all file causes an image of the batch compiler to be saved in the file upto.all. This image contains the symbol table information for all the modules in the compiler. It can be used to recompile individual files in the compiler after they have been modified (but beware of changes in signatures).

Cross compiling

If you have, say, the Vax object files in mldist/mo.vax and need to build the sparc object files, then you can build a batch compiler to generate sparc object files by using the -target option of makeml:

```
% makeml -vax bsd -target sparc
```

(the -target option implies the -batch option). Then the resulting batch compiler can be used to generate the sparc mo files by executing (in directory src):

```
smlc < all
```

Then make a directory mldist/mo.sparc and move the new object files into this directory. These object files can then be used to bootstrap the compiler on a sparc machine in the usual way.