# SML.NET User Guide

Andrew Kennedy, Claudio Russo, Nick Benton
Microsoft Research Ltd.
Cambridge, U.K.

V1.2 build 1613 of Friday, 02 June 2006

## Contents

# 1 Introduction

SML.NET is a Standard ML compiler [2, 3] for the .NET Common Language Runtime. Its features are:

**Support for all of Standard ML**. SML.NET compiles all of SML '97 [2] except for some minor omissions documented in Appendix A.

**Support for the Basis library**. Almost all of the Standard ML Basis Library [1] is implemented. Omissions and discrepancies are listed in Appendix B.

**Command-line compilation**. SML.NET supports traditional compilation from the command-line.

**Interactive compilation environment**. Alternatively, you can control the compiler from an interactive environment. This lets you set and query options incrementally and to see the signatures of compiled and imported SML modules.

**Support for Visual Studio .NET**. This distribution includes an optional, experimental package for Visual Studio .NET that allows you to edit, build and debug SML.NET projects from within the development environment; see Section 5 for an overview and additional installation instructions.

**Automatic dependency analysis**. In either mode of compilation, the compiler requires only the names of root modules and a place to look for source code. It then does dependency analysis to determine which files are required and which need recompilation.

**Produces verifiable CLR IL**. The output of the compiler is verifiable MSIL (Microsoft Intermediate Language) for the CLR.

**Smooth interop with other languages on the CLR**. SML.NET extends the Standard ML language to support safe, convenient use of the .NET Framework libraries and code written in other languages for the CLR. SML.NET can both consume and produce .NET classes, interfaces, delegates, etc. These extensions are discussed in full in Section 4.

**Whole program optimization**. SML.NET performs optimizations on a whole program (or library) at once. It usually produces small executables with fairly good performance.[1]

Its limitations are:

[1] Though this is an early release and the performance variation is very wide. Compared with SML/NJ, for example, some real programs go four times faster and some go ten times slower.

**No interactive evaluation**. The interactive environment is for compilation of stand-alone applications or libraries only. SML expressions can not be evaluated interactively and the `use` command is not available.[2]

**Whole program optimization**. Top-level SML modules are not compiled individually to .NET object code. Instead, some compilation takes place on separate modules (type checking, translation to the compiler's own intermediate form, and some optimizations) but most is deferred until *after* the linking together of top-level modules. This improves performance of the generated code, but significantly increases (re)compilation times. To give a rough idea, it takes a couple of minutes to recompile a 25,000 line SML application on a 1.33GHz Athlon with 512MB of RAM.

**Only CLR types at boundaries of compiled code**. The exposed interfaces of applications or DLLs compiled by SML.NET may only refer to CLR types (classes, interfaces, delegates, etc.). They may not expose SML-specific types (functions, datatypes, records, etc.). In particular, this restriction means that one cannot compile an arbitrary SML module into a DLL for consumption even by other SML.NET programs: the module must be either linked into the client program at compile-time or use only CLR types at its interface.

## 1.1 About this document

This guide is aimed at programmers already familiar with SML. The textbook by Paulson [3] is an up-to-date introduction to SML'97.

Section 2 describes requirements, installation, and takes you through the compilation of a simple program. Section 3 presents the compilation environment. Section 4 describes extensions to Standard ML for interfacing to CLR libraries and for implementing new CLR classes inside SML. Section 5 documents the Visual Studio .NET support package.

[2] For programs that make no use of the language extensions it is possible to develop and test them using a compiler such as Moscow ML or Standard ML of New Jersey and then to use SML.NET to produce final executables.

## 1.2 Licence



## 1.3 Credits

The SML.NET team is Nick Benton, Andrew Kennedy and Claudio Russo. The compiler also includes code by George Russell. We would like to thank all the others who have contributed code, support, web space, demos, test cases, gripes and bug reports both to SML.NET and to its precursor, MLj: Gavin Bierman, Tom Chothia, Stephen Gilmore, Dave Halls, Bruce McAdam, Chris Reade, John Reppy, Peter Sestoft, the SML/NJ developers, Ian Stark, Audrey Tan, Team PLClub, Bent Thomsen, Lone Thomsen, Christian Urban.

## 1.4 Mailing list

A mailing list for SML.NET users has been set up. To join the list, send mail to `smlnet-users-request@jiscmail.ac.uk`. To view the archives of the list, see http://www.jiscmail.ac.uk/lists/smlnet-users.html.

# 2 Getting started

## 2.1 Requirements

**Microsoft Windows**. You can use any of Windows 98, ME, NT4.0, 2000, XP Home or XP Professional to run the compiler. We generally recommend Windows 2000 or XP, and some of the server-side parts of .NET require the Server or Professional versions of those operating systems.

**.NET Framework**. Available from `http://msdn.microsoft.com/net`. The *Redist* includes the runtime, libraries and tools (notably an assembler) required to run SML.NET. The *SDK* includes full documentation, samples, and other tools (e.g. a disassembler and stand-alone verifier) which you may find useful. Visual Studio .NET is also useful if you wish to write parts of your applications in C♯ or VB. In particular, you can design user interfaces graphically and then link the autogenerated code with SML.NET code.

**An editor**. The SML.NET compiler is not an integrated development environment, so, unless you install the optional support for Visual Studio .NET (Section 5), you will need a text editor. Typically you will run the editor concurrently with the SML.NET compilation environment.

**SML/NJ compiler (optional)**. The SML.NET compiler was developed using Standard ML of New Jersey (SML/NJ). If you wish to build the SML.NET compiler from sources then you will need SML/NJ version 110, obtainable from `http://cm.bell-labs.com/cm/cs/what/smlnj`.

## 2.2 Installation

To install SML.NET, simply unpack the distribution `smlnet.tar.gz` to obtain a directory `smlnet`. The compiler can be run directly from the `bin` subdirectory but for convenience you may want to extend `PATH` with this directory.
**IMPORTANT:** To avoid problems, please unpack your distribution to a path that does not contain spaces, eg. `C:\smlnet` but not `C:\Program Files\smlnet`. We hope to lift this restriction in a future release.

On Windows 98 and Windows ME, if the compiler is installed somewhere other than `C:\smlnet` then you must also set `SMLNETPATH` to that directory.

## 2.3 Configuration (Optional)

SML.NET is designed to support multiple versions of the .NET Framework, as determined by the values of two environment variables: `FrameworkDir` and `FrameworkVersion`. If these variables are undefined, SML.NET will infer their values by running the helper program `bin\getsysdir.exe`. When defined, `FrameworkDir` should specify the machine's installation directory for all versions of the .NET Framework, and `FrameworkVersion` the particular version of the .NET Framework that you wish to target (`FrameworkVersion` must name a subdirectory of `FrameworkDir`.) Typical settings might be:

```
set FrameworkDir=C:\WINDOWS\Microsoft.NET\Framework
set FrameworkVersion=v1.1.4322
```

**WARNING:** It is strongly recommended that you `clean` your project whenever you change to a different version of the .NET Framework (See Section 3.14).
**NB:** the `Framework` environment variables are pre-defined when you start a **Visual Studio .NET 200*x* Command Prompt**, so running SML.NET from the shell will automatically target the correct .NET Framework for that version of Visual Studio.

## 2.4 Example: Quicksort

To test its operation on a demonstration program, run SML.NET in command-line mode as shown below:

```
C:\smlnet>bin\smlnet -source:demos\sort Sort
SML.NET 1.0
Analysing dependencies...done.
...
Linking modules...done.
Compiling whole program...done.
Compilation succeeded: output in Sort.exe
C:\smlnet>
```

This tells SML.NET to compile a top-level SML structure called `Sort`, assumed to be found in the directory specified (`demos\sort`) in a file called `Sort.sml`, and to produce an executable with the name `Sort.exe`. If this is the first time that any SML.NET program has been compiled, then you will notice that most of the Basis library is compiled too.

Alternatively, first enter the interactive compilation environment:

```
C:\smlnet>bin\smlnet
SML.NET 1.0
\
```

Then type the commands that follow the `\` prompts below:

```
\ source demos\sort
\ export Sort
\ make
Analysing dependencies...done.
Linking modules...done.
Compiling whole program...done.
Compilation succeeded: output in Sort.exe
\ run 20
Before sorting: 60 37 14 91 68 45 22 99 76 53 30 7 84 61 ... 23
After sorting: 7 14 15 22 23 30 37 38 45 46 53 60 61 68 ...  99
\ quit
C:\smlnet>
```

The `source` command tells the compiler where to look for SML source files, which by default have the extensions `.sml` (for structures and functors) and `.sig` (for signatures). The `export` command specifies the name of a top-level SML structure to be exported as a .NET class in the executable. Then `make`

tells SML.NET to compile and link the program. The compiler will put the output in a file `Sort.exe`, and the assembler source in a file `Sort.il`. Finally `run` executes the program with the arguments specified.

Just to prove that we really have compiled a self-contained .NET program, from a command prompt type

```
C:\smlnet>Sort 10

Before sorting: 30 7 84 61 38 15 92 69 46 23
After sorting: 7 15 23 30 38 46 61 69 84 92

C:\smlnet>
```

If you have installed the .NET Framework SDK you can type-check the output using the verifier and look at it using the disassembler:

```
C:\smlnet>peverify Sort.exe

Microsoft (R) .NET Framework PE Verifier
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

All Classes and Methods in Sort.exe Verified

C:\smlnet>ildasm Sort.exe
```

Most of the classes listed by `ildasm` were generated by SML.NET to implement SML code, but `Sort` is special, as it was listed explicitly as a parameter to `export`. At its simplest, `export` is followed by a comma-separated list of top-level SML structures, each of which will be *exported* as a class with the same name. The signature of each structure determines what will appear in the class. Functions are exported as static methods, and other values are exported as static read-only fields. There are strong restrictions on the types of functions and values that can be exported, described in detail in Section 4.

For this example, there is just one exported function, called `main`, which by convention identifies the entry point for an executable, as is the case in C♯.

## 2.5 Demonstration programs

The distribution comes with a few more demos. Explore the `demos` subdirectory for details and documentation.

# 3 Compiling programs

In this section we discuss in detail the compilation environment provided by SML.NET.

Within the environment you can type `help` or just `?` to get a list of available commands. For details on a particular command, type `help` *command*.

## 3.1 Command syntax

In general a command consists of an alphabetic keyword optionally followed by parameters separated by commas. The case of keywords is ignored. Parameters representing file or directory names must be enclosed in quotes if they include spaces.

Most commands set or extend 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.

## 3.2 Mapping of module identifiers to files

A multiple module SML.NET program usually consists of a collection of top-level SML structures and signatures, each stored in a separate file. In contrast to other compilation managers, SML.NET 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 (using `export`) and a means of mapping SML signature, structure and functor identifiers onto file names, SML.NET 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`, each functor *funid* into a file called *funid*`.sml`, and each signature *sigid* into a file called *sigid*`.sig`. Users of Moscow ML will be familiar with this pattern. The compiler then just starts its dependency analysis from the root structures specified using the `export` command.

**PITFALL:** SML.NET will not infer dependencies that are not there in the source. Thus if your project includes file `State.sml`, containing

```
structure State = struct
  val state = ref (NONE: string option)
end
```

file `Effect.sml`, containing

```
structure Effect = struct
  val _ = State.state := SOME "initialised!"
end
```

and file `Main.sml`, containing

```
structure Main = struct
  fun main() = case !State.state of NONE => 1 | SOME s => (print s;0)
end
```

then exporting `Main` alone will drag in `State`, but will *not* drag in the purely side-effecting initialisation code in `Effect.sml`. Note that `Main` does not reference `Effect` (either directly or indirectly via the referenced entity `State`). To force the evaluation of `Effect`, you *must* add an explicit reference to the `Effect` structure, for instance by inserting a vacuous `local open Effect in end` declaration to `Main`:

```
structure Main = struct
  local open Effect in end
  fun main() = case !State.state of NONE => 1 | SOME s => (print s;0)
end
```

## 3.3 Specifying a path for sources

`source`

By default, the compiler looks for source files in the working directory. The command

`source` *directory* `,` ... `,` *directory*

specifies a comma-separated list of directories to search for source files. The directory names are normalized with respect to the directory current at the time `source` is executed, and if `source` is already set then new directories are appended. To display the current path type `source?`, and to restore the default behaviour (using the working directory) type `source`.

## 3.4 User-defined mappings

`structure, signature, functor`

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, functor or signature identifier to filename. The commands

`structure` *strid*=*filename* `,` ... `,` *strid*=*filename*
`signature` *sigid*=*filename* `,` ... `,` *sigid*=*filename*
`functor` *funid*=*filename* `,` ... `,` *funid*=*filename*

let you do this. The filenames are normalized with respect to the directory current when the command is issued, and repeated use of the commands extend the existing mapping.

Multiple entities can be mapped to the same file, in which case the file must contain top-level module bindings for each of those entities.

To find out what file a particular module maps to (whether through explicit mapping or the source path and default naming convention), use the commands:

`structure` *strid*`?`
`signature` *sigid*`?`
`functor` *funid*`?`

## 3.5 Recompilation

`make,remake`

When a source file is changed and `make` or `remake` is invoked, SML.NET 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*). If a recompiled structure matches an unchanged signature then no modules which depend on it will be recompiled.

Command `make` compiles the current project, but links and assembles only if the sources have changed or the target does not exist.

Command `remake` compiles the current project, but re-links and re-assembles even when the sources have not changed and the target already exists.

## 3.6 Exporting classes to .NET

`export, main`

The `export` command indicates to the compiler which top-level SML structures are to be exposed as public classes in the compiled executable or library. The syntax of `export` is the following:

`export` *strid*⟨`=`*classname*⟩ `,` ... `,` *strid*⟨`=`*classname*⟩

By default, structures are exported as top-level classes with names matching that of the SML.NET structure. For example, the command `export S, M` generates two public classes `S` and `M`. These names can be overridden, typically to introduce a namespace. For example,

```
export S=MyLib.Class1, M=MyLib.Class2
```

will produce two classes `Class1` and `Class2` in namespace `MyLib`.

Executables must have an entry point. By default, the compiler looks for an exported structure with a function called `main` or `Main` with one of the following types:

```
val main :  unit -> unit
val main :  unit -> int
val main :  string option array option -> unit
val main :  string option array option -> int
```

These types correspond to those accepted by the C♯ compiler for `main`.

## 3.7 Importing classes from .NET

`reference, lib`

In order to import classes from .NET libraries and code written in other languages it is necessary to name the assemblies (roughly speaking, DLLs) that are required. This is done using the `reference` command:

`reference` *libname*`,` ... `,` *libname*

The `lib` command:

`lib` *directory* `,` ... `,` *directory*

allows a path for searching for `.dll`s to be specified. SML.NET follows the same rules as C♯ in searching for assemblies:

1. The working directory.
2. The .NET system directory.
3. The directories set by the `lib` command.

4. The directories set in the `LIB` environment variable.

Many useful system assemblies are referenced by the `config.smlnet` script run from the `bin` directory when SML.NET starts up. If you prefer not to reference all of these, the minimal set of references actually required to compile structures in the Basis is `mscorlib.dll` and `System.dll`.

## 3.8 Output

`out, target, run, log`

The compiler can be instructed to produce an executable (`.exe`) or a shared library (`.dll`) with the following commands:

```
target library
target exe
```

The default is to produce executables, in which case the compiler expects to find some exported structure containing a suitable `main` function.

By default, the name of the output is *X*`.exe`, for executables whose `main` function is in structure *X*, or *X*`.dll` for DLLs whose first exported structure is *X*. This name can be overridden by typing `out` *name*. The syntax `out` *X*`.exe` and `out` *X*`.dll` combines the action of `target` and `out` in a single command.

To run a successfully-compiled executable from within the compilation environment, type

`run` *args*

where *args* are the arguments passed to the executable.

You can log the compiler messages to a file by typing `log` *filename*. To turn logging off just type `log`.

## 3.9 Shell commands

`cmd, cd`

From within the compilation environment you can issue shell commands using the syntax

`cmd` *command*

It is also possible to change the working directory from within the environment:

`cd` *directory*

The syntax `cd?` queries its current value.

## 3.10 Printing types and signatures

**`type`**

The command

`type` *longid*

provides type and signature information for any entities with the name *longid*. These entities may be value bindings, types, structures or exceptions and may come from external .NET .DLLs, the SML Basis or the current project. `type` provides a primitive, but extremely useful, form of online documentation for use during program development. It is particularly convenient for checking how SML.NET views libraries written in other languages.

## 3.11 Options

`on, off, ?`

There are a large number of compiler options available. They are turned on and off using `on` *option* and `off` *option* and can be queried using *option*`?`. (Alternative syntax: *option*`+` or just *option* to enable, and *option*`-` to disable). We document the most useful options here:

`debug` Emit debugging information (default: off). If enabled, some symbol and line number information is generated in the assembler file, and the `/debug` option is passed onto the .NET assembler.

`debug.exnlocs` Exception line number information (default: on). If enabled, the compiler will insert appropriate code into the output so that when SML exceptions appear at top level the runtime reports the SML structure and line number where the exception was raised.

`debug.symFromLoc` Emit truncated source expressions as the symbolic names for their values in debug builds (default: off). This can be useful for viewing the values of intermediate expressions when debugging.

`sml.seqwithtype` Sequential withtype declaration (default: off). The Standard requires that multiple bindings in the `withtype` clause associated with `datatype` declarations are interpreted simultaneously. It is arguably more useful to interpret them sequentially, as does SML/NJ.

`warn.value` 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 A.3 for more details).

`warn.match` Non-exhaustive match warnings (default: on). These are displayed when a `case` or `handle` does not handle all possible values and so could result in the `Match` exception being thrown.

`warn.bind` Non-exhaustive bind warnings (default: on). These are displayed when a `val` binding can result in the `Bind` exception being thrown.

`verbose` Verbose compiler messages (default: off).

## 3.12 Additional assembler options

`ilasm`

It is sometimes necessary, or just convenient, to be able to pass extra command-line options to the assembler. The `ilasm` command takes a comma separated list of additional options with their values, if any. For instance:

```
ilasm CLOCK, KEY=keyFile.snk
```

instructs `ilasm.exe` to report timings and generate a *signed* assembly using the key pair in file *keyFile.snk* when next invoked by the compiler (see the `ilasm.exe` documentation). These settings correspond to the concrete shell command `ilasm.exe ... /CLOCK /KEY=keyFile.snk ...`. Note that the extra options are passed to ilasm in addition to any options already implied by other SML.NET settings. The commands `ilasm` and `ilasm?` reset and query the current options.

## 3.13 Avoiding stack overflow

By default, under Windows, CLR executables receive a fixed, and fairly low, ceiling on the amount of stack space they can use during execution. This can pose a problem for some functional programs that rely heavily on deep, non-tail recursive call patterns.

Fortunately, the Windows system utility `editbin.exe` can be used to modify an executable so that it demands a higher ceiling for the stack. For example

```
editbin /stack:100000000 Sort.exe
```

sets a (huge) ceiling of ca. 100MB. See the `editbin` documentation for more details.

## 3.14 Cleaning up

The SML.NET compiler will cache information (assembly metadata, sml object files and dependency information) in sub-directory `bin` of the distribution and subdirectories `.smlnetdep` and `.smlnetobj` of each directory on the source path.

To delete this information, use the `clean.bat` script; `clean` scrubs the basis directories, everything cached in `bin` and in the current directory and its subdirectories. Thus executing `clean` from `C:>\smlnet` should return the distribution to a pristine state.

Use this script when you suspect the compiler has got itself confused (rare, but possible). With this release, we do not recommend you have more than one SML.NET process going at the same time, especially if the projects under compilation share source code other than the basis.

## 3.15 Command files and command-line operation `@`

A sequence of compiler commands can be collected together in a file *name*`.smlnet` and then executed simply by typing `@`*name*. Inside *name*`.smlnet` the commands must be separated by newlines. A single command can be split over several lines provided that the splits occur following commas. This is useful for commands that specify lists of items.

The special command file `config.smlnet`, located in the compiler `bin` directory, is interpreted when the compiler is run. It is a useful place to set options appropriate to all projects.

It is also possible to execute compiler commands directly from the command line, either before entering the compilation environment (to set options such as `source`, for example) or without entering the environment at all (to build an executable, for example). Simply precede the command names with hypens or slashes, and separate the command from its arguments by a colon instead of a space. Identifiers at the end of the line are interpreted as inputs to an implicit `make`; if any `make` occurs at all then the compilation environment is not entered. For example,

```
C:\smlnet>smlnet -source:demos\sort Sort
```

will compile our quicksort example to produce a file `Sort.exe`, and

```
C:\smlnet>smlnet -verbose
```

will enter the compilation environment in verbose mode.

## 3.16 Summary of commands

`help` *command*
`help:`*command*
`?` *command*
`?:`*command*
`help`
`?`
Displays command list or more detailed help on a particular command.

`type` *longid*
`type:`*longid*
Displays type signatures for entities with name *longid* (values, types, structures, and signatures).

`source` *directory* , ... , *directory*
`source:`*directory* , ... , *directory*
`source?`
`source`
Extends, queries or resets path in which source files are searched for.

`structure` *strid=filename* , ... , *strid=filename*
`structure:`*strid=filename* , ... , *strid=filename*
`structure` *strid*`?`
`structure:`*strid*`?`
`structure?`
`structure`
Extends, queries or resets mapping of structures to files.

`signature` *sigid=filename* , ... , *sigid=filename*
`signature:`*sigid=filename* , ... , *sigid=filename*
`signature` *sigid*`?`
`signature:`*sigid*`?`
`signature?`
`signature`
Extends, queries or resets mapping of signatures to files.

`functor` *funid=filename* , ... , *funid=filename*
`functor:`*funid=filename* , ... , *funid=filename*
`functor` *funid*`?`
`functor:`*funid*`?`
`functor?`
`functor`
Extends, queries or resets mapping of functors to files.

`make`
Compile the current project, linking and assembling only if the sources have changed or the target does not exist.

`remake`

Recompile the current project, linking and assembling even if the sources have not changed.

`export` *strid*⟨`=`*classname*⟩`,` ... `,` *strid*⟨`=`*classname*⟩
`export:`*strid*⟨`=`*classname*⟩`,` ... `,` *strid*⟨`=`*classname*⟩
`export?`
`export`

Extends, queries or resets the set of structures to be exported as .NET classes.

`reference` *libname*`,` ... `,` *libname*
`reference:`*libname*`,` ... `,` *libname*
`reference?`
`reference`

Extends, queries or resets the set of external .NET libraries (`.dll`s) referenced by the current project.

`lib` *directory* `,` ... `,` *directory*
`lib:`*directory* `,` ... `,` *directory*
`lib?`
`lib`

Extends, queries or resets path in which `.dll`s are searched for.

`target exe`
`target:exe`
`target library`
`target:library`
`target?`

Sets or queries whether current project is an executable or a library.

`out` *name*
`out:`*name*
`out` *name*`.exe`
`out:`*name*`.exe`
`out` *name*`.dll`
`out:`*name*`.dll`
`out?`
`out`

Sets, queries or resets file name for compiler output and, optionally, project type.

`run` *args*

Runs the current project (if executable and built) with specified command-line arguments.

`cmd` *command*
`cmd:`*command*

Runs the specified shell command.

`cd` *directory*
`cd:`*directory*
`cd?`

Sets or queries the current working directory.

`log` *filename*
`log:`*filename*
`log`
Sets compilation log file and starts logging, or stops logging.

`on` *option*
`on:`*option*
*option*`+`
*option*
`off` *option*
`off:`*option*
*option*`-`
*option*`?`
Where *option* is one of `debug`, `debug.exnlocs`, `debug.symfromloc`, `sml.seqwithtype`, `warn.value`, `warn.match`, `warn.bind`, `verbose`. Sets, resets or queries the associated compiler flag.

`ilasm` *option*⟨`=`*value*⟩ `,` ... `,` *option*⟨`=`*value*⟩
`ilasm:`*option*⟨`=`*value*⟩ `,` ... `,` *option*⟨`=`*value*⟩
`ilasm?`
`ilasm`
Extends, queries or resets the additioanl options passed to the .NET assembler, `ilasm`. Useful for signing assemblies, adding resources such as icons, etc.

`@`*name*
Executes commands from file *name*`.smlnet`.

## 3.17 Trouble with the .NET IL assembler `ilasm.exe`?

There is a bug in the 1.0 and 1.1 versions of `ilasm.exe`, the IL assembler used by SML.NET: the debug information in an IL method can be associated with at most one source file. In SML.NET target code, due to inlining, a target method will often be associated with more than one source file. To avoid the assembler bug, SML.NET prefers to use its own version of the assembler in `bin\ilasm.exe`. If, for some reason, this private `ilasm.exe` does not work on your machine you can simply delete it from the `bin` directory. In that case, SML.NET will use the `ilasm` from your .NET Framework installation. In addition, as a less satisfactory workaround, in debug mode, SML.NET will produce a dummy source file (with extension `*.ml`) containing the concatenation of all sources referenced from the executable. It is this dummy file, not the original code, that is referenced by the debugger and should be used for setting breakpoints (but this can be rather confusing to the user).

The executable `bin\ilasm.exe` is Copyright ©2002-2003 Microsoft Research Ltd. and is covered by a separate licence in file `MSR-EULA`. If you are not happy with the terms of this licence, simple delete the executable `bin\ilasm.exe`.

# 4 Language extensions for .NET

A significant aspect of the SML.NET compiler is its support for seamless interoperation with other .NET languages and libraries. The approach taken is the following:

- Where possible, map .NET features into equivalent SML features. For example, static methods are mapped to top-level function bindings in SML.
- Where there is no obvious equivalent, extend the SML language. For example, new .NET classes can be defined within SML using a new `classtype` construct.

The sections which follow describe the extensions in detail. Each section is accompanied by small example programs illustrating a single feature. These live in directories displayed on the right of the section heading.

## 4.1 Namespaces, classes and nesting

`samples\structures`

If one ignores the class hierarchy and instance fields and methods (i.e. a non-object-oriented fragment of .NET), then .NET classes can be seen (and are used) as a minimal module system, providing a way of packaging together static fields, methods, and nested classes that are logically related. Namespaces in .NET provide a further level of structuring, grouping together many classes into a single group. We model both using the SML module system.

Top-level namespaces (e.g. `System`) are reflected in SML.NET as top-level structures, with nested namespaces (e.g. `System.Drawing`) reflected as substructures (e.g. structure `Drawing` inside structure `System`).

Classes are reflected in SML.NET as three separate bindings:

- as type identifiers (see Section 4.2.2),
- as values of function type used to construct instances of the class (see Section 4.3.1), and
- as structures containing value bindings that reflect static fields and methods (see Sections 4.4.1 and 4.4.2) and substructure bindings for nested classes.

For example, within the namespace `System.Threading` (reflected as a structure `Threading` inside a top-level structure `System`) the class `Mutex` is mapped to an SML type identifier `Mutex`, to a value identifier `Mutex`, and to a structure `Mutex`.

Namespaces and classes interpreted as structures can be manipulated like any other structure in SML: they can be rebound, constrained by a signature[3], passed to functors, and opened.

Opening of namespaces-as-structures through `open` is analogous to C♯'s `using` construct. However, when used with classes-as-structures the `open` mechanism is more powerful, permitting unqualified access to static fields and methods. Also, nested namespaces become visible as structures.

[3] This is not supported in the initial release

| .NET type | C♯ type | SML.NET type |
|---|---|---|
| System.Boolean | bool | bool |
| System.Byte | byte | Word8.word |
| System.Char | char | char |
| System.Double | double | real |
| System.Single | float | Real32.real |
| System.Int32 | int | int |
| System.Int64 | long | Int64.int |
| System.Int16 | short | Int16.int |
| System.SByte | sbyte | Int8.int |
| System.String | string | string |
| System.UInt16 | ushort | Word16.word |
| System.UInt32 | uint | word |
| System.UInt64 | ulong | Word64.word |
| System.Exception | System.Exception | exn |
| System.Object | object | object |

Table 1: Correspondence between types in .NET and SML.NET

## 4.2 Types

### 4.2.1 Built-in types

`samples\builtintypes`

Table 1 lists .NET types (in C♯ notation and as fully-qualified .NET type names) that have direct equivalents in SML.NET. Amongst .NET types only the usual SML Basis types have equality status and can be passed as arguments to SML's polymorphic equality operator `=`, that is all those in Table 1 except for `real`, `Real32.real` and `exn`.

### 4.2.2 Named .NET types

`samples\namedtypes`

Any named .NET type (class, value type, enumeration, interface or delegate) can be referred to from within SML.NET using the same syntax as in C♯. This syntax works because of the interpretation of .NET namespaces as nested structures, discussed above. For example:

```
type XMLParser = string -> System.Xml.XmlDocument
```

### 4.2.3 Array types

`samples\arrays`

Single-dimensional .NET arrays behave almost exactly like SML arrays: their size is fixed at time of creation, indexing starts at zero, equality is based on identity not value, and an exception is raised on out-of-bounds access or update. Therefore the SML type constructor `array` corresponds to .NET's type constructor `[]`. (But see Section 4.2.4 for a discussion of null-valued arrays).

The .NET exception `System.IndexOutOfRangeException` corresponds to the SML exception `Subscript`.

Some, but not all, array types inherit from the class `System.Array`, so you can invoke methods on values of these array type and cast their array values

to and from class types. The restriction is that the element type must be an *interop input type* in the sense of Section 4.2.5.

### 4.2.4 Null values

`samples\option`

In .NET, variables with class or array types (known collectively as *reference* types) are allowed to take on the value `null` in addition to object or array instances. Operations such as method invocation, field access and update, and array access and update, raise `NullReferenceException` if their main operand is `null`.

SML does not have this notion, and values must be bound explicitly when created. Thus operations such as assignment, indirection, and array access and update are inherently safer than the corresponding operations on .NET. We wished to retain this safety in our extensions to SML, and so interpret a value of .NET reference type as "non-null instance".

Nevertheless, when a .NET field of reference type is accessed from SML or a value of reference type is returned from an external method invoked by SML, it may have the value `null` and this must be dealt with by the SML code. Also, it should be possible to pass null values to .NET methods and to update .NET fields with the null value. Fortunately the SML basis library already defines a type that suits this purpose perfectly:

```
datatype 'a option = NONE | SOME of 'a
```

The `valOf` function (of type `'a option -> 'a`) can be used to extract the underlying value, raising `Option` when passed `NONE`.

We interpret values of .NET reference type that cross the border between SML and .NET as values of an `option` type. For example, the method `Join` in class `System.String` has the C♯ prototype:

```
public static string Join(string separator, string[] value);
```

This maps to an SML function with signature

```
val Join : string option*string option array option->string option
```

### 4.2.5 Interop types

We will use the term *interop type* for the types described in the previous sections and new .NET types defined inside SML.NET code. Interop types can be used in SML extensions such as casts, overloading, implicit coercions, and (with additional restrictions) in exported structures.

A special case of an interop type is an interop *input* type. Input types describe values that might be passed *into* SML.NET from external .NET classes, and therefore must assume the possibility of null values for reference types.

To be precise, an interop type is one of the following:

1. A .NET value type (primitive or struct). Examples: `int`, `System.DateTime`.

2. A .NET class, interface or delegate type defined externally or from within SML.NET code by `_classtype` or `_interfacetype` (see Sections 4.8.1 and 4.8.3). Examples: `string`, `System.IEnumerable`, `System.EventHandler`.
3. An array whose element type is an interop *input* type. Examples: `string option array`, `System.DateTime array`.
4. Possibly-null versions of either of the above. Examples: `string option array option`, `System.EventHandler option array option`.

A type is an interop *input* type if it is one of (1) or (4) above.

## 4.3 Objects

### 4.3.1 Creating objects

`samples\new`

In C♯, instances of a class are created using the syntax

$$\texttt{new}\ class\texttt{(}arg_1, \ldots, arg_n\texttt{)}$$

where $arg_i$ are the arguments to one of the constructors defined by the class.

We avoid the need for any new syntax in SML.NET by binding the class name itself to the constructor function. If there is more than one constructor, then the binding is overloaded (see Section 4.4.3).

Constructors can be used as first-class values, and implicit coercions are applied using the same rules as for methods (see below). For example:

```
val fonts =
  map System.Drawing.Font [("Times", 10.0), ("Garamond", 12.0)]
```

### 4.3.2 Creating and invoking delegate objects

`samples\delegates`

.NET and C♯ support a kind of first-class function called a *delegate*. A delegate object is an instance of a named delegate type that wraps up a method and (in the case of instance methods) its context. Delegate types are mapped to two SML bindings: the type itself, just as with class types, and a function binding for the delegate constructor, which takes an SML function as its only argument. For example, suppose that the following delegate type was declared in C♯:

```
public delegate int BinaryOp(int x, int y);
```

This is reflected as an SML type `BinaryOp` and a function with signature

```
val BinaryOp : ((int*int)->int) -> BinaryOp
```

used to construct delegate objects from SML functions. For example:

```
val adder = BinaryOp(op+)
```

To actually apply a delegate to some arguments, you simple use its (virtual) `Invoke` method:

```
val 3 = adder.#Invoke(1,2)
```

Every delegate has an implicitly defined `Invoke` method (C♯ actually employs syntactic sugar that calls this method under the hood).

Delegates are reference types with base class `System.MulticastDelegate`. Thus they come equipped with a suite of methods, beyond the crucial `Invoke` method discussed here (see the .NET documentation).

### 4.3.3 Casts and cast patterns

`samples\casts`

A new syntax is introduced to denote C♯-style casts:

*exp* `:>` *ty*

It can be used to cast an object up to a superclass:

```
open System.Drawing
val c = SolidBrush(System.Color.get_Red()) :> Brush
```

Explicit coercions are sometimes required when passing .NET objects to SML functions and constructors, as coercions are only applied implicitly when invoking .NET methods.

The same syntax can also be used to cast an object down to a subclass, with `System.InvalidCastException` thrown if the actual class of the object is not compatible. A safer alternative that combines downcasting with C♯'s `is` construct is the use of `:>` inside SML patterns:

*pat* `:>` *ty*

This can be used to provide a construct similar to type-case found in some languages. For example, this code from `demos\xq\Xmlinterop.sml` switches on the type of an `XmlNode`:

```
fun nodetoxmldata (n : XmlNode) =
 case n of
   elem :> XmlElement =>
    let val SOME name = elem.#get_Name()
        val first = elem.#get_FirstChild()
        val children = gather (first, [])
    in SOME (Elem(name, List.mapPartial nodetoxmldata children))
    end

 | data :> XmlCharacterData =>
    let val SOME s = data.#get_Data()
    in SOME (C(stringtoscalar s))
    end

 | _ => NONE
```

The pattern *id* `:>` *ty* matches only when the examined expression has the class type *ty*, in which case the identifier *id* is bound to the expression casted down to type *ty*.

Cast patterns can be used like any other pattern. They can appear in `val` bindings, as in

```
val x :> System.Windows.Forms.Window = y
```

to give an effect similar to downcasting in expressions but raising SML's `Bind` exception when the match fails. They can also be used in exception handlers, such as

```
val result = (f y)
  handle e :> System.Security.SecurityException => 0
```

in order to handle (and possibly deconstruct) .NET exceptions. The *order* in which handlers appear is important. In the example below, the exception type `DivideByZeroException` is a subclass of `ArithmeticException` so if the handlers were switched the second handler would never be reached.

```
fun test x = (do_some_stuff x)
  handle y :> ArithmeticException => f y
       | _ :> DivideByZeroException => g x
```

Finally, the behaviour of C♯'s `e is c` can be emulated by

```
case e of _ :> c => true | _ => false
```

## 4.4 Fields, methods and properties

Static (per-class) fields and methods are mapped to value and function bindings in SML located in the structure corresponding to their class. For example, the `PI` static field in the `System.Math` class, accessed from C♯ using `System.Math.PI`, maps to a value binding for `PI` in the SML structure `System.Math` accessed using the same syntax. Likewise, the `Cos` static method in the same class is mapped to a value binding of `Cos` in the structure `System.Math`.

Non-static (instance) fields and methods are handled specially through new syntax for field access and method invocation:

*exp*`.#`*name*

Here *exp* is an SML expression with a .NET object type, and *name* is the name of an instance field or method.

Properties are really just C♯ syntactic sugar formalizing the commonplace "get/set" design pattern. No special support is provided in SML.NET, so they must be accessed through their underlying methods which have the names `get_`*P* and `set_`*P* for a property called *P*.

We now describe how field and method types are mapped into SML.

### 4.4.1 Fields

`samples\fields`

Immutable .NET fields (`readonly` and `const` in C♯) are given types as explained in Section 4.2, using `option` to denote the possibility of null values for objects or arrays. For example, a field declared in C♯ using

```
public static string language_name = "C#";
```

is interpreted as having type `string option`.

For the most part, mutable fields can be treated as if they had SML `ref` types: they can be deferenced using `!` and assigned to using `:=`. So fields declared by

```
class C {
  public static int counter; // static
  public int size; // instance
...
}
```

can be used as if it is an SML reference cell whose contents have type `int`. For example:

```
C.counter := !C.counter + 1
...
fun size (x:C) = !(x.#size)
```

In fact, mutable fields are given special types that are a generalization of Standard ML's `ref` type constructor (see Section 4.7.)

Every mutable field also gives rise to a *type* binding of the same name. You should think of this as a new kind of SML reference type; in reality, it just abbreviates a particular storage type describing this kind of field (see Section 4.7.)

```
val counterRef : C.counter = C.counter
val c : C = ...
val sizeRef : C.size = c.#size
```

### 4.4.2 Methods

`samples\methods`

.NET method types are interpreted as follows. First, `void` methods are considered as having `unit` result type; similarly methods that take zero arguments have `unit` argument type. Second, .NET supports multiple arguments directly but SML does not, so methods with multiple arguments are given a single tuple argument type. Finally, when arguments and results are objects or arrays, their types are interpreted using the `option` type constructor as described earlier.

Consider the following method from class `System.String`:

```
public static string[] Split(char[] separator, int count);
```

Its type is interpreted as

```
val Split : char array option * int -> string option array option
```

and can be called using an ordinary function application:

```
fun split(c:char array,i:int) = valOf(System.String.Split(SOME c, i))
```

Here is an example of instance method invocation:

```
(* Create an object *)
val xmldoc = XmlDocument()
(* Invoke an instance method on it *)
val _ = xmldoc.#Load(filename)
```

### 4.4.3 Overloading and implicit coercions

`samples\overloading`

.NET permits the *overloading* of methods: the definition of multiple methods with the same name within a single class. The methods are distinguished by their argument types. Furthermore, C♯ and other languages support implicit coercion of arguments in method invocations. The combination of these features can lead to ambiguity, which C♯ resolves statically by picking the *most specific method* with respect to an ordering on argument types, rejecting a program if there is no unique such method.

SML.NET allows implicit coercions on method invocation using C♯'s reference widening coercions together with an additional coercion from $T$ to $T$ `option` for any .NET reference type $T$. We do not allow C♯'s numeric widening coercions to be implicit as the 'spirit of SML' is to use explicit conversions such as `Int64.fromInt` for these.

We do not allow ambiguity to be resolved by C♯-style most specific method rules, as these interact unpleasantly with type inference: our intention is to have typing rules and an inference algorithm such that a program is accepted iff there is a unique resolution of all the method invocations (with respect to the rules). Use of the 'most specific' rule during inference can lead to type variables becoming bound, and hence ambiguities far from the point of the rule's application being resolved in unexpected ways.

## 4.5 Value Types

`samples\valuetypes`

.NET provides support for an extensible set of unboxed, structured values called value types (C♯'s *structs*). Consider the C♯ declaration of lightweight, integer pairs:

```
public struct Pair {
  public int x;  /*  mutable! */
  public int y;  /*  mutable! */
  /* constructor */
  public Pair(int i,int j){x = i; y = j;}
  /* functional swap, returns a new pair */
  public Pair swap(){return new Pair(y,x);}
  /* destructive swap, modifies 'this' pair */
  public void invert(){int t; t = x; x = y; y = t; return;}
}
```

Value types, like ordinary classes, can have fields and instance methods. However, because value types are sealed (cannot be subclassed), they do not need to be boxed on the heap (otherwise used to provide uniform representations); nor do they need to carry run-time type descriptors (used to support checked downcasts and virtual method invocation). For the most part, you can view instances of value types as structured primitive types. Indeed, primitive type such as `int` *are* value types. Every value type derives from the base class `System.ValueType`. Value types are passed by value, or copying semantics, not by reference.

In SML.NET, you can use the same syntax for accessing fields and invoking methods on a value of value type as for objects of reference types. So for a

pair `p`, `p.#x, !(p.#x), p.#swap()` are all legal. Method invocation works at primitive types too, eg. `1.#ToString()`.

In SML.NET, invoking on a value type first copies the value, takes the address of the copy and then passes this address as the "this" argument of the method. Similary, accessing a mutable field of some value first copies that value. This ensures that values are effectively *immutable* in SML.NET, a basic requirement for the implementation of a functional language. Thus `p.#invert()` has no effect on the value of `p`, nor does `let val r = p.#x in r := 1 end`. The expressions do have side-effects, but they cannot alter `p`.

Some value types do have mutable semantics, perhaps employing mutable instance fields that are updated by instance methods. Our copying semantics implies that these updates cannot be observed (since they mutate a temporary copy of the value, not the original value itself). To cater for such types, SML.NET supports an alternative invocation semantics. In addition to invoking directly on a value, SML.NET lets you invoke on any kind of SML.NET *reference* to a value type. Thus, `(ref p).#invert()` and, more usefully,

```
let val r = ref (Pair(1,2)) in r.#invert();!r end,
```

which returns the modified value `Pair(2,1)`, are both legal. This works as expected for fields too:

```
let val r = ref (Pair(1,2)) in (r.#x) := 3;!r end
```

returns the modified value `Pair(3,2)`.

This mechanism applies to all of the storage kinds of SML.NET (see Section 4.7). By accessing fields or invoking methods on expressions of type `(`*ty*`,`*kind*`)reference`, you can, if required, mutate: static fields of classes, mutable instance fields of objects and value types and even arbitrary addresses.

### 4.5.1 Boxing and unboxing

.NET provides a uniform object model, allowing any value of value type to be viewed at type `object` (and thus stored in collection classes, etc.). In detail, every value type has a corresponding *boxed* representation of proper class type `System.ValueType` and, by subtyping, `System.Object`.

*Boxing* a value allocates a new, appropriately tagged, object on the heap and copies that value into the object. In SML.NET, the boxed form of a value type is obtained by an *upcast*, eg. `p:>System.Object`, to some suitable CLR reference type. The supertype is typically `System.Object`, but may be `System.ValueType`. Boxing casts work for primitive types too, eg. `1:>object`.

*Unboxing* an object extracts a value from a heap allocated object (but requires a dynamic type check). In SML.NET this is achieved by a *downcast* from some object type to a value type. For instance, for `obj : System.Object`, the expressions `obj:>Pair` and `obj:> int` unbox the underlying value, returning a value. Note that, like any other downcast, these involve a dynamic test and can fail, raising `System.InvalidCastException`.

### 4.5.2 Null values

In SML.NET, every non-primitive value type has an implictly defined `null` value, bound in the structure of that same name (eq. `Pair.null` is equivalent to

`Pair(0,0)`); The null value is provided in case the value type has no associated .NET constructor: it can be used to initialise a reference prior to setting up its state. For example:

```
let val r = ref Pair.null
in
    (r.#x) := 1;
    (r.#y) := 2;
    !r
end
```

returns the value `Pair(1,2)`, but without calling a constructor.

## 4.6 Enumeration Types

`samples\enums`

In .NET, an enumeration type is a distinct value type, declared with a set of named constants of that type. Every enumeration derives from proper class type `System.Enum`. Each enumeration type has an underlying (signed or unsigned) integral type, equivalent to one of the SML.NET types `Int`*N*`.int` or `Word`*N*`.word` for $N \in \{8, 16, 32, 64\}$.

In SML.NET, a .NET enumeration type is imported as a pseudo datatype of the same name. The datatype has a single, unary constructor, named after the type, that constructs an enum from a value of the underlying type. The named constants of the enumeration are imported as equivalently named constant constructors, abbreviating particular applications of the proper constructor. The derived constructors are bound in a separate structure, named after the enumeration type. The proper constructor and its derived constant constructors can be used in patterns as well as expressions.

For example, values of the C♯ enumeration type:

```
public enum MyEnum { A , B , C = A }
```

may be manipulated in SML.NET as follows:

```
type enum = MyEnum
fun fromInt i = MyEnum i
fun toInt (MyEnum i) = i
fun toString MyEnum.A = "A"
  | toString MyEnum.B = "B"
  | toString MyEnum.C = "C" (* this case is redundant ! *)
  | toString (MyEnum i) = Int.toString i
fun fromString "A" = MyEnum.A
  | fromString "B" = MyEnum.B
  | fromString "C" = MyEnum.C
  | fromString s = MyEnum (valOf (Int.fromString s))
```

Whether in a pattern or expression, the constructor `MyEnum.A` is completely equivalent to writing `MyEnum 0` instead.

Enumerations, like other primitive types, are also value types: their values can be cast to and from `System.Enum`, `System.ValueType` and `System.Object`, and can also have methods invoked on them.

Note that, in .NET, the constants defined for an enumeration type are not necessarily exhaustive of the possible range of values, nor are they required to name distinct values. This feature distinguishes .NET enumeration types from the familiar SML pattern of defining an $n$-valued enumeration by declaring a datatype with $n$ distinct, constant constructors.

## 4.7 Storage Types

`samples\references`

Standard ML has a minimalist type system that supports only one kind of mutable storage cell, *ty* `ref`, with the following interface:

```
type ty ref
val ref      :  ty -> ty ref
val !        :  ty ref -> ty
val :=       :  (ty ref * ty) -> unit
```

Operationally, the expression `ref` *exp* evaluates *exp* of type *ty* to some value *v*, allocates a new object containing a single field storing *v* on the heap and returns a handle to that object of type *ty* `ref`; `!` and `:=` read and write the contents of the field. References have identity and may be tested for equality using SML's equality predicate `= :(''a * ''a) -> bool`.

SML.NET compiles values of type *ty* `ref` to an instance of a private class with a single mutable instance field of type *ty*. However, the CLR provides a much wider range of mutable structures: static fields of classes, instance fields of heap allocated multi-field objects and instance fields of stack allocated value types. The CLR even provides a general address type to support call-by-reference parameter passing (C♯'s `ref` and `out` parameters). For interoperation with .NET libraries, SML.NET has to support these kinds of storage as well.

In SML.NET, all of these types are described as particular instantiations of a more general type constructor, `(`*ty*`,`*kind*`) reference`. Unlike SML's `ref` type constructor, this type constructor takes *two* type parameters. The first simply describes the type of the value stored in the cell. The second parameter is a pseudo (or phantom) type that identifies the particular *kind* of storage cell. The kind of a storage cell describes its physical representation and thus the precise runtime instructions needed to implement reads and writes.

The advantage of introducing a parametric notion of storage cell, indexed by kind, is that it allows SML.NET to treat `!` and `:=` as generic operations, polymorphic, as in ML, in the contents of a storage cell but also polymorphic in the physical representation of that cell. To achieve this, we generalise the Standard ML types of these operations to the following *kind*-polymorphic types in SML.NET:

```
val !    :  (ty,kind) reference -> ty
val :=   :  ((ty,kind) reference * ty) -> unit
```

In practice, this means that you can use the same familiar notation (`!` and `:=`) to manipulate all kinds of storage cells, even though they compile to rather different runtime instructions.

### 4.7.1 Storage kinds

The *kind* parameter of a storage cell of type `(`*ty*`,`*kind*`) reference` can take one of the following forms:

**heap:** used for ordinary Standard ML references, this kind describes a single-field, ML allocated object. Standard ML's primitive *ty* `ref` type constructor is defined as an abbreviation for the SML.NET reference type:

```
type ty ref  =  (ty,heap) reference
```

Thus Standard ML's `ref` *function*, that allocates a new `ref` cell, actually has type:

```
val ref  :  ty -> ty ref
         =  ty -> (ty,heap) reference
```

**(*classty*,*fieldname*)static:** used for a static field, this kind describes a static field by the name of the class and the name of the field.

**(*classty*,*fieldname*)field:** used for an instance field, this kind describes the superclass of the object *classty* in which the field *fieldname* is defined (the sort of the class is enough to identify whether this is a field of heap allocated .NET object or a field of a .NET value type (C♯ "struct" type).

**address:** used to describe the address of a storage cell. An address can point to the interior of an ML ref, to a static field, a stack-allocated local variable (e.g. from C♯), an instance field of a heap allocated object or an inlined value type. Addresses are an abstraction of all of the above storage types.

Bear in mind that storage cells of different kinds have distinct types and are thus *not* type compatible, e.g. it is a static type error to create a list containing both an integer heap reference and an integer static field reference (because the kinds of the references are not unifiable).

Address kinds are used to describe the type of C♯ call-by-reference `ref` and `out` parameters, and typically occur in the (argument) types of imported methods. Because an address can refer to a value on the stack, whose lifetime is limited to that of its enclosing stack frame, values of address type and instance fields of value types are subject to certain data-flow restrictions, described in detail in Section 4.7.3.

### 4.7.2 Address operators (&)

It is possible to take the *address* of any kind of storage cell (including another address reference) using the SML.NET primitives:

```
type ty &  =  (ty,address) reference
val &      :  (ty,kind) reference -> ty &
           =  (ty,kind) reference -> (ty,address) reference
```

The `&` function allows you to pass any kind of SML.NET storage cell (including an SML `ref`) to a C♯ method expecting a call-by-reference parameter (marked as `ref` or `out` in C♯) by taking, and then passing, the address of that cell.

For instance, given the C♯ swap function:

```
public static void swap(ref int i, ref int j){
  int temp = i; i = j; j = temp; return;
}
```

which is imported with SML.NET type:

```
val swap : int & * int & -> unit
```

Then we can swap the contents of two (ML) references as follows:

```
val (ra,rb) = (ref 2,ref 3)
val _ = swap(& ra,& rb)
val (ref 3,ref 2) = (ra,rb)
```

Operationally, taking the address of a storage cell returns the address of the particular field storing its contents; taking the address of an address is simply the identity.

### 4.7.3 Byref types

The various kinds of storage in SML.NET fall into two categories: storage cells whose representations are first-class values of the CLR and storage cells whose representations are second-class CLR *addresses*. References of kind `address` and, less obviously, `(`*`classty`*`,`*`fieldname`*`)field`, where *classty* is a value type (not a proper class type), belong to this second category. In SML.NET, these are collectively called *byref* types.

In the CLR, an address can, amongst other things, be the address of a value allocated on the call-stack (e.g. an imported address of C♯ local variable), or the address of a field of a value on the stack (e.g. an imported address of a field of a C♯ local variable). Such an address is only valid for the lifetime of the stack frame from which it was taken. To prevent such ephemeral addresses from being read or written outside their lifetime (when they are unsafe dangling pointers), the CLR imposes restrictions on values of address type: they cannot be stored in static fields or instance fields of CLR values or objects [4]. Because SML.NET's byref storage types compile to CLR addresses, which must obey the CLR's rules, byref values and types can only be used in limited ways. In detail, SML.NET imposes the following restrictions:

- A value of byref type must not occur as a free variable of any function or class declared within its scope (this includes anonymous functions).
- A byref type cannot be used as the argument type of a datatype or exception constructor.
- A tuple (or record) cannot contain a field of byref type, unless that tuple is the immediate argument to a method or constructor.
- A structure expression may not export a value of byref type.
- A function may take a value of byref type as an argument, but not a tuple containing a component of byref type.
- A method may take a single byref as an argument, and may also take a tuple containing a component of byref type.

[4] Aside from these restrictions, addresses *can* be passed to other methods, supporting Pascal-style call-by-reference passing of local variables.

- A function or method may not return values of byref type.
- When matching a structure to a signature, the implementation of any opaque type in that signature may not be a byref type.
- A byref type may not be used as the type of an argument to an SML.NET class constructor or as the type of any value bound in the optional local declaration of an SML.NET class declaration. This restriction will be lifted in a later release of the SML.NET compiler.
- Finally, to ensure the above properties are preserved by SML's type instantiation of polymorphic values and parameterised type constructors, type arguments, whether explicit or inferred, cannot be byref types, nor may they be *kinds* describing byref types. Exceptions to this rule: `!`, `:=` and `&` (which are known to be safe) may be instantiated at kinds describing byrefs. [5]

## 4.8 Defining new .NET types

### 4.8.1 Class declarations

`samples\classtype`

The mechanisms described so far give the SML programmer access to .NET libraries, but they do not support the creation of new class libraries, nor do they allow for the specialisation of existing .NET classes with new methods coded in SML. For this we introduce a new construct whose syntax is shown below.

| | | |
|---|---|---|
| *dec* | ::= | `_classtype` ⟨`{`*attributes*$_1$`}`⟩⟨`[`*cmod*`]` ⟨`{`*attributes*$_2$`}`⟩⟩ |
| | | *class-name pat* ⟨`:` *superdecs*⟩ |
| | | `with` ⟨`local` *dec* `in`⟩ ⟨ *methoddecs* ⟩ `end` |
| *superdec* | ::= | *ty* \| *ty exp* |
| *superdecs* | ::= | *superdec* \| *superdec* `,` *superdecs* |
| *methoddec* | ::= | ⟨`{`*attributes*`}`⟩⟨`[`*mmod*`]`⟩*method-name pat* `=` *exp* |
| | \| | ⟨`{`*attributes*`}`⟩⟨`[`*mmod*`]`⟩*method-name* : *ty* |
| *methoddecs* | ::= | *methoddec* \| *methoddec* `and` *methoddecs* |

This introduces a new class type *class-name* defined by the following elements:

- The optional, and typically absent, *attributes*$_1$ and *attributes*$_2$ list any class or constructor attributes (see Section 4.9). [6]
- The optional class modifier in *cmod* can be `abstract` or `sealed` and has the same meaning as in C♯.

[5] In SML.NET, the values `!`, `:=` and `&` should not be rebound by the user, but this is currently *not* enforced by the compiler.

[6] You must explicitly write a (possibly) empty `[`*cmod*`]` phrase to indicate any constructor attributes; this avoids ambiguities in the grammar.

- The expression *class-name pat* acts as a 'constructor header', with *pat* specifying the formal argument (or tuple of arguments) to the constructor. Any variables bound in *pat* are available throughout the remainder of the class type construct.

  Unlike C♯, multiple constructors are *not* supported; a future enhancement might allow additional constructors to be expressed as invocations of a 'principal' constructor.

- The optional *superdecs* specifies the superclass that *class-name* extends and any interfaces that it implements. The superclass clause (which must occur first, if at all) contains an argument (or tuple of arguments) *exp* to pass to the superclass (*ty*) constructor. If absent, this defaults to `System.Object()`. The remaining types in a *superdecs* clause must be (distinct) interfaces.

- *dec* is a set of SML declarations that are local to a single instance of the class.

- The optional *methoddecs* is a simultaneous binding of instance method declarations, defined using a syntax similar to that of ordinary functions, but with optional qualifiers `final` and `protected` preceding the method identifiers. An abstract method is declared by omitting is implementation, but declaring its type. An abstract method may have an optional, explicit `abstract` qualifier. Note that methods may have attributes and that abstract methods must have function types.

  The class definition is bound within its own body, allowing recursive references via `this`; the methods of a class can have types that mention the class type itself. [7]

In keeping with tradition, and to demonstrate that classes are usable in SML.NET without reference to .NET libraries, Figure 1 presents a variation on the classic coloured-point example.

Notice the absence of any direct support for field declarations. Instead, the declarations following `local` are evaluated when a class instance is created but are accessible from the method declarations for the lifetime of the object. In this example we have mimicked private mutable fields using `ref` bindings (`x` and `y`), with initial values provided by arguments to the constructor (`xinit` and `yinit`). The methods, which may be mutually recursive (as suggested by the `and` separator), can refer both to these arguments and to the bindings introduced by `local`.

The `ColouredPoint` class derives from the `Point` class, passing two of its constructor arguments straight on to its superclass constructor. It has no local declarations and a new method that simply returns its colour.

Because the declarations are local to the class *instance*, it is not possible to gain access to the corresponding declarations for other instances of the class. In C♯, private fields for other instances *can* be accessed directly, for example, to implement an `equals` method. In SML.NET, this can be emulated by providing appropriate 'get' and 'set' methods for the fields.

[7] Direct mutual recursion between class declarations is not yet supported in this release, but can be simulated using forward declarations of abstract classes.

```
structure PointStr =
struct
  _classtype Point(xinit, yinit)
  with local
    val x = ref xinit
    val y = ref yinit
  in
        getX () = !x
    and getY () = !y
    and move (xinc,yinc) = (x := !x+xinc; y := !y+yinc)
    and moveHoriz xinc = this.#move (xinc, 0)
    and moveVert yinc = this.#move (0, yinc)
  end

  _classtype ColouredPoint(x, y, c) : Point(x, y)
  with
        getColour () = c : System.Drawing.Color
    and move (xinc, yinc) = this.##move (xinc*2, yinc*2)
  end
end
```

Figure 1: Coloured points in SML.NET

The special identifier `this` has the same meaning as in C♯, referring to the object on which a method was invoked. It is used in `Point` to define horizontal and vertical movement using the more general `move` method.

The C♯ language provides a syntax (`base`) which allows a method overridden by a subclass to invoke the method that it is overriding. Instead, we provide a syntax

*exp*`.##`*method-name*

that can be used only within a class definition on objects of that same class, and means "invoke method *method-name* in the superclass, ignoring any overriding of the method in the current class". It is used in `ColouredPoint` to redefine `move` using the `move` method defined in `Point`, making coloured points "faster movers" than plain points. By the magic of virtual method dispatch, the `moveHoriz` and `moveVert` inherited by coloured points also inherit this speed increase.

As mentioned in Section 4.4.3, .NET allows overloading of methods. We support this in `_classtype` declarations in order to extend existing .NET classes that include overloaded methods. No special syntax is required: the method name is simply repeated in separate declarations, as in the example below:

```
_classtype C ()
with
      m(x:int) = ...process ints...
  and m(x:string option) = ...process Strings...
end
```

```
functor Wrapper(type T) =
struct
  _classtype W(x : T)
  with
    get() = x
  end
  fun wrap   (x : T) = W(x)
  fun unwrap (w : W) = w.#get()
end

structure IntListWrapper = Wrapper(type T = int list)
structure IntFunWrapper = Wrapper(type T = int->int)
```

Figure 2: Using functors

### 4.8.2 Class types and functors

`samples\classfunctor`

At present, the CLR does not support parametric polymorphism. We therefore restrict the types of methods in classes to be *monomorphic*. However, using SML's functor construct, it is still possible to parameterise classes on types and values. Figure 2 gives an example.

When applied to a particular type `T`, the functor provides a new class type `W` and functions `wrap` and `unwrap` that convert values between `T` and `W`. The class types `IntListWrapper.W` and `IntFunWrapper.W` can then be used to pass around objects that wrap up SML values of type `int list` and `int->int`. (If one wished to use these wrapper classes to, for example, store SML values in .NET collections, one would also have to include a hash function and equality test in the class.)

### 4.8.3 Interface declarations

`samples\interfaces`

SML.NET users may declare their own interface types. The syntax of interface declarations is similar to that of classtype declarations, but more restricted. An interface type must have no constructor, no local declarations, no superdec declaring a superclass and base constructor, and no concrete method implementations:

$$
\begin{array}{rl}
\textit{dec} ::= & \texttt{\_interfacetype}\ \langle\texttt{\{}\textit{attributes}\texttt{\}}\rangle \\
& \textit{interface-name}\langle\texttt{:}\textit{superdecs}\rangle \\
& \texttt{with}\ \ \langle\ \textit{methoddecs}\ \rangle\ \texttt{end} \\
\textit{superdec} ::= & \textit{ty} \\
\textit{superdecs} ::= & \textit{superdec}\ \mid\ \textit{superdec}\ \texttt{,}\ \textit{superdecs} \\
\textit{methoddec} ::= & \langle\texttt{\{}\textit{attributes}\texttt{\}}\rangle\langle\texttt{[}\textit{mmod}\texttt{]}\rangle\textit{method-name}\ \texttt{:}\ \textit{ty} \\
\textit{methoddecs} ::= & \textit{methoddec}\ \mid\ \textit{methoddec}\ \texttt{and}\ \textit{methoddecs}
\end{array}
$$

The methods of an interface type, but not its super declarations, may make recursive references to that interface type. [8]

[8] Direct mutual recursion between interface declarations is not yet supported in this release, but it can be simulated using forward declarations of interfaces.

### 4.8.4 Delegate declarations

`samples\delegates`

SML.NET users may declare their own delegate types. The syntax of delegate declarations uses a concise form of classtype declaration:

*dec* ::= `_classtype` ⟨`{`*attributes*`}`⟩
*delegate-name* `of` *ty*

- The optional, and typically absent, *attributes* lists any class attributes (see Section 4.9).
- The type argument *ty* of a delegate must be an SML function type.

A delegate declaration declares the type *delegate-name*; the delegate constructor *delegate-name* of (higher-order) type *ty* `->` `classname`; and implicitly declares an `Invoke` instance method of the (function) type *ty*.

For example, the `BinaryOp` delegate class of Section 4.3.2 may be declared within SML.NET as:

```
_classtype BinaryOp of (int*int)->int
```

This simultaneously declares the type constructor `BinaryOp`, the (higher-order) class constructor `val BinaryOp: ((int*int) -> int) -> BinaryOp` and the implicit `Invoke` method of type `(int*int)-> int`.

Unlike ordinary SML function types, delegates can be exported in the sense of Section 4.10, providing a rudimentary way of inter-operating at the level of first-class functions.

## 4.9 Custom Attributes

`samples\attributes`

SML.NET, like C♯, enables programmers to use and declare new forms of custom meta-data using *attribute classes*. Programmers can annotate SML.NET code with *instances* of attribute classes, whether these classes were imported or declared within SML.NET itself. These attribute values may be retrieved at run-time using reflection.

The standard example is that a framework might define a `HelpAttribute` attribute class that can be placed on certain program elements (such as classes and methods) to provide documentation displayed in meta-data aware class browsers.

Here's a simple attribute class defined in C♯.

```
public class CSharpAttribute : System.Attribute {
    public CSharpAttribute(){}
    public string Property {... set{...};}
    public string Field = null;
};
```

In SML.NET, a new attribute class is defined by declaring a class that extends `System.Attribute`, eg:

```
_classtype MLAttribute(s:string option):System.Attribute()
with ToString () = s end;
```

Attributes can be attached to certain, but by no means all, declarations in an SML.NET program using attribute expressions.

An SML.NET attribute expression is an essentially static description of the data required to construct the corresponding instance of the attribute class when demanded to do so by reflection. Syntactically, an attribute expression is simply an application of an instance constructor of some attribute class to an $n$-tuple of constant arguments ($n \geq 0$). For instance, we might annotate another class, `MLClass()`, as follows:

```
_classtype {MLAttribute("this is a class attribute"),
            MLAttribute("this is another class attribute")}
           [] {MLAttribute("this is a constructor attribute")}
           MLClass()
with
    {MLAttribute("this is a method attribute")} method () = ()
end
```

An attribute expression may be further modified by a sequence of (mutable) field and property initialisers, again supplied with constant values. The initialisers are executed in order, just after the instance is constructed by reflection.

For instance, attribute instances of the above `CSharpAttribute` type may be qualified as follows:

```
CSharpAttribute() where Field = SOME "field-value" end,
CSharpAttribute() where Property(SOME "property-value") end,
CSharpAttribute() where
   Field = SOME "field-value",
   Property(SOME "property-value")
end,
(*NB: in named arguments, we use the name of the property,
  not the name of the set method *)
```

Ignoring any type annotations used to resolve overloading, constant values must be literals or values immediately constructed from literals, of the following SML.NET types[9]: `string` (and `string option`), `char`, `bool`, `Word8.word`, `Int16.int`, `int`, `Int64.int`, `Real32.real`, `real`, any imported enumeration type, or `System.Type`[10]

The precise grammar of attribute expressions is given below:

$$
\begin{array}{rcl}
\mathit{attributes} & ::= & \texttt{\{}\langle \mathit{attexpseq} \rangle\texttt{\}} \\
\mathit{attexp} & ::= & \mathit{exp}\ \langle\ \texttt{where}\ \mathit{namedargs}\ \texttt{end}\ \rangle \\
\mathit{attexpseq} & ::= & \mathit{attexp}\ \mid\ \mathit{attexp}\ \texttt{,}\ \mathit{attexpseq} \\
\mathit{namedarg} & ::= & \mathit{fieldname}\ \texttt{=}\ \mathit{exp}\ \mid\ \mathit{propertyname}\ \mathit{exp} \\
\mathit{namedargs} & ::= & \mathit{namedarg}\ \mid\ \mathit{namedarg}\ \texttt{,}\ \mathit{namedargs}
\end{array}
$$

[9] The supported attribute argument types are the ones required by the CLS (.NET's Common Language Subset).

[10] `System.Type` arguments are not supported in this release.

Imported attribute classes will typically have named as well as positional parameters but attribute classes declared in SML.NET can only ever have positional parameters, corresponding to the arguments of the (sole) class constructor. This is simply a consequence of our design: SML.NET's `_classtype` declarations cannot explicitly declare named fields or properties.

In the current release, attribute expressions may only be placed on classes, class constructors, methods, interfaces, interface methods, delegate classes and delegate constructors. C♯ provides rather more places for attributes, most notably assembly and parameter attributes; if there is sufficient demand, we may extend SML.NET's coverage further.

**PITFALL:** C♯ assumes all attribute class names have the suffix `Attribute` and allows the programmer to omit the suffix from attribute expressions; In SML.NET, we make no such assumption, but require the full name of the attribute class in attribute expressions.

## 4.10 Exporting structures

By default, all SML.NET declarations are private to the generated executable or DLL. In order to make declarations available outside – even simply to expose an entry point – it is necessary to *export* selected top-level structures using the compiler's `export` command.

In the current version of the compiler it is only possible to export structures whose signature can be mapped directly back to .NET types. The following rules are applied for export:

- Top-level SML structures are exported as .NET classes, possibly in some namespace if specified by the `export` command.
- Value bindings with function type are exported as static methods. A function type is exportable if
  - Its result type is an exportable type.
  - Its argument type is either `unit`, an exportable *input* type, or a tuple of exportable input types. An argument type, or tuple component, may also be the address of an exportable input type (akin to C♯'s `ref` and `out` parameters).
- Value bindings with exportable types are exported as static read-only fields.
- Class, interface and delegate type declarations are exported as .NET (nested) classes, if:
  - The type of the constructor argument is either `unit`, an exportable input type, or a tuple of exportable input types (as for functions).
  - The methods all have exportable types (as for functions).
- A type is an *exportable type* if it is an interop type (see Section 4.2.5) that does make not use of non-exported class, interface or delegate types. A type is an *exportable input type* if it is an interop input type (see Section 4.2.5) that does not make use of non-exported class, interface or delegate types.

- All other bindings are non-exportable.

Here is a contrived example:

```
structure S = struct
  _classtype Counter(init) with
    local val privateRef = ref init
    in
        current () = !privateRef
    and setToCurrent (i:int &) = i := (!privateRef)
  end
  val x = 5
  val y = "I'm a string"
  val c = Counter(5)
  fun f(x) = x+1
  fun prnl(x) = case x of NONE => () | SOME s => print (s ^ "\n")
end
```

This will be exported as a .NET class `S` with the following C♯ pseudo-signature:

```
public sealed class S {
  public class Counter {
    public Counter(int init);
    public virtual int current();
    public virtual void setToCurrent(ref int i);
  }
  public static readonly int x;
  public static readonly string y;
  public static readonly C c;
  public static int f(int x);
  public static void prln(string x);
}
```

A future release of SML.NET will support the export of arbitrarily *nested* structures, merging classes and substructures with equivalent names on export. This will provide a convenient way to define classes with both instance and static methods, thus mirroring the semantics of class import.

# 5 Visual Studio .NET Support

Depending on your choice of download, your distribution may include an experimental Visual Studio .NET package for SML.NET. This package extends the Visual Studio Development Environment with SML specific editor and project support. Installation of this package is optional and at your own risk.

The SML editor provides code colouring, bracket matching, syntax checking, Intellisense on both .NET and SML.NET libraries and even interactive type inference [11]. An SML.NET project is just a C++ Makefile project that invokes the SML.NET batch compiler. It can participate in Visual Studio multi-project solutions. The package includes simple wizards for creating new SML.NET projects and project items. As usual, a project may be built in a *Debug* or

[11] All implemented in a pure SML.NET, masquerading as a classic COM component.

*Release* configuration. Debug executables may be run under the Visual Studio source code debugger. Although by no means perfect, the debug experience has improved substantially over SML.NET 1.1 and is worth another visit.

## 5.1 Licence

The Visual Studio package is Copyright ©2002-2003 Microsoft Research Ltd. and is covered by a separate licence in file `MSR-EULA`. Please take the time to read the licence before installing the package.

## 5.2 Requirements

The package requies the .NET Framework 1.0 or 1.1 (1.1 is recommended for debugging), and an edition of Visual Studio .NET 2002, 2003 or 2005 (otherwise known as versions 7.0, 7.1 and 8.0) that includes C++ support, such as Professional or Academic editions. The Express editions of Visual Studio .NET 2005 are, unfortunately, not suitable.

## 5.3 Installation

To install SML.NET for Visual Studio, simply run the executable `install.exe` included in the distribution. By default, it installs the SML.NET support, using the highest version of the CLR, for the highest version of VS currently installed on your machine. Run `install /help` for more options that allow you to override the inferred defaults. To uninstall, run `uninstall.bat` or just `install.exe /u`.

## 5.4 Working In Visual Studio

This section is a very brief guide to working with SML.NET projects in Visual Studio .NET.

### 5.4.1 Opening an existing Project

The distribution comes with a sample Visual Studio Solution. This is a graphical, multi-language Game Of Life application: the user interface is written in C♯ (project **Client** producing executable `Client.exe`) but the Game Of Life algorithm is written in SML.NET (project **Server** producing library `Server.dll`).

Start VS and open the sample solution file `Life2002.sln`, `Life2003.sln`, or `Life2005.sln` (as appropriate for your version of VS) from the directory `vs\demo\Life` in your distribution. You should be able to compile the solution out of the box, choose **Build→Solution**. To *run* the solution, you may need to first manually set the startup project to be the C# **Client** project (For some reason, VS sometimes assumes that the startup project is the **Server** library, but this is a `.dll` and cannot be run). One way to set the startup project: in the **Solution Explorer Window**, right-click on the **Client** and select **Set as Startup Project**. Now choose **Debug→Start** to run the application. You should see a small **Life** application window appear; in the application, select **File→Go** to start the animation.

After running the project, try editing file `Server.sml` in the **Server** sources. The SML code should appear in colour in the editor window. If not, something

went wrong with your installation. Edit the code; any syntax errors will be underlined with red squigglies (and displayed in the Task List). After you have have built the project at least once, you will be able to use Intellisense on SML identifiers, see type and build errors in the Task List window, set breakpoints in the margin of source code, etc. Note that Intellisense will only work if the source file participated in the most recent build. Also, Intellisense will only work against libraries referenced in the most recent build.

The file `Server.sml` contains some comments suggesting things to try in the Visual Studio editor. See the Visual Studio Documentation for more details on using the Visual Studio Editor.

### 5.4.2 Creating a new Project

Start Visual Studio .NET. Choose **File→New→Project ...**. In the **New Project** dialog box, supply a *name* and location for your new project. In the **Project Types** pane, open the folder **SML.NET Projects**. In the **Templates** pane, double-click on the **SML.NET Application** icon. (The other icon selects a library, or `.dll`, project). This should create the new project.

Now select **View→Solution Explorer** You should see a new tab with an SML.NET project (the icon actually says it is a C++ project, but never mind). Click on the *name*`.sml` file in the **Solution Explorer** tab. If it appears in the editor with coloured syntax, all is well.

You can now build (**Build→Build Project**) and run (**Debug→Start**) the project. When run, the skeleton application does nothing but exit.

New source files may be added to the project by choosing **Project→Add New Item ...**, providing a name for the file, expanding the **Visual C++** tree[12], selecting the **SML.NET** subtree, and then selecting the appropriate **SML Template**.

Every new project includes an SML.NET script, `script.smlnet`, as a project item. This script is passed to the batch compiler in both debug and release builds and should be used to set any additional compiler options, like referencing additional assemblies, including additional SML source directories, etc.

For example, to reference the assemblies in files $filename_1$, ..., $filename_n$ add the following line to `script.smlnet`:
`reference` $filename_1$ `,...,` $filename_n$
(Under VS2003 and higher, the project tree will also contain a **References** node, but it is completely ignored by SML.NET and should not be used to reference assemblies.)

To include additional SML source in directories $dirname_1$, ..., $dirname_n$ add the following line to file `script.smlnet`:
`source` $dirname_1$ `,...,` $dirname_n$

Each project also includes with a `README.html` file with some hints and handy links to the online SML Basis documentation (which you can browse within VS) and the online SML.NET manual (also included in the distribution).

Note that you can easily customize the project **Build**, **Rebuild** and **Clean** actions by editing the project's properties; select the **NMake** property subtree and edit the **NMake** command line properties.

[12] SML.NET projects masquerade as C++ makefile projects.

### 5.4.3 Debugging

SML.NET is able to produce symbolic debug information for use by the Visual Studio Debugger (and other CLR debuggers such as the free source code debugger available in the .NET SDK).

For some entertainment, open a Life solution, select the **Debug** configuration (**Build→Configuration Manager . . . →Debug**), and build the solution again (see Section 5.4.1). Now set a breakpoint by clicking the grey margin in some SML file (some good locations are marked `(*BREAK*)` in file `Server.sml`). Run the application by choosing **Debug→Start**. After selecting **File→Go**, the application should break into the VS source code debugger. The call stack will probably contain a mixture of C♯ and SML.NET stack frames. Clicking on stack frames will take you to the relevant source code. You can even step through the program, including stepping through SML.NET source code and inspecting SML.NET bindings (in their raw, compiled form) on the stack.

Although the situation has improved, the SML.NET compiler does not always produce *enough* debug information: control flow, data representations, identifier names and source locations are frequently altered or obscured by the various optimisations in the compiler. Even so, the information that does remain can still be useful in tracking down bugs.

From version 1.2 of the compiler, bindings in the **Locals** window now enter and exit scope appropriately. The values of (truncated) sub-expressions, not just identifiers, may be reported as locals in the **Locals** window, provided you rebuild your sources with compiler option `on debug.symFromLoc` (this is off by default).

In debug builds, constructed values now have a symbolic tag field derived from the constructor name. SML.NET stack frames displayed in the **Call Stack** will mostly have meaningful, not mangled, source names.

As a convenience, values of heap-allocated SML types (such as tuples, records and non-flat datatypes) now support additional diagnostic `ToString()` and `ToString(int depth)` virtual methods that can be invoked in the VS **Immediate** window to inspect their values at runtime (these methods cannot be called from SML.NET source code.)

Building in a **Debug** configuration is equivalent to setting the `debug` flag of the command-line SML.NET compiler.
**WARNING: Debug** configurations inhibit many SML.NET optimisations and produce considerably worse compiled code than **Release** configurations.
**NOTE:** Your debug experience is affected by the IL assembler used by your installation of SML.NET (see Section 3.17 for more details).

## 5.5 Customizing the Package Installer

The package installer may fail because it cannot locate a tool it requires, so we have included its SML.NET source code. Should you need to fix the installer, the code resides in `vs\install.sml` and `vs\RegTools.sml`, with build script `vs\install.smlnet`. After editing the installer, you can rebuild it as follows.

```
cd vs;..\bin\smlnet @install.smlnet
```

Note the installer's executable is placed in the root directory of the distribution (not in `vs`). The installer must reside in this directory to work correctly,

as it uses its own location to infer the location of the compiler binaries.

# A Language restrictions

## A.1 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.*`). A correct implementation in CLR 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.

## A.2 Non-uniform datatypes

The SML.NET 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.

This restriction will be lifted in a future release of the compiler.

## A.3 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* [2, Section 4.7]). SML.NET 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 xand it can be argued that it is an omission from the Definition.

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

## A.4 Overloading

SML.NET 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 [2, Appendix E]. The following typechecks under SML/NJ and Moscow ML but not under SML.NET 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 SML.NET will use the largest context permitted.

# B The Standard ML Basis Library

A large part of the Standard ML Basis Library is implemented. Online documentation for the basis library may be found at http://www.standardml.org/Basis/index.html.

Table 2 lists the structures that exist with omissions and discrepancies detailed (to appear).

# C Support for ML-Lex, ML-Yacc and SML/NJ Libraries

Many people have asked about using SML.NET with some of the extra tools and libraries which come with SML/NJ. This is usually not hard, but does involve a certain amount of work writing .smlnet scripts to define entity mappings. To avoid everybody having to do this independently, we have included SML.NETified versions of some of the most popular.

The distribution comes with SML.NET versions of the parser generator tools ML-Lex and ML-Yacc. The support libraries for ML-Yacc are in directory `smlnet\lib\parsing` and if you wish to incorporate code which has been generated by ML-Yacc into your project you simply have to place this directory on the SML.NET source path *and* execute the `smlnet\lib\parsing\sources.smlnet` script to map the required module names to the files in this directory (see the xq demo for an example of how to do this). The ML-Lex and ML-Yacc tools themselves can be found in the demos directory, though if you already have the SML/NJ versions, there's no particular advantage in running the .NET executable versions.

Separately, the `smlnet\lib\smlnj-lib.smlnet` script predefines a mapping from module names to file names that lets you easily use some of the SML/NJ utility libraries (typically under `sml\src\smlnj-lib\Util`, but the precise location depends on your SML/NJ installation). This mapping is incomplete in the current release.

# References

[1] E. R. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library reference manual*. Cambridge University Press, to appear. In preparation, online reference available at `http://www.standardml.org/Basis/index.html`.

| Structure | Omissions and discrepancies |
|---|---|
| top-level | no `use` |
| `Array` | none |
| `BinIO` | no `getPos`*, `setPos`*, no functional I/O |
| `Bool` | none |
| `BoolVector`,`BoolArray` | none |
| `Char`=`WideChar` | `Char.maxOrd` = 65535 |
| `CharVector`=`WideCharVector` | none |
| `CharArray`=`WideCharArray` | none |
| `CommandLine` | none |
| `Date` | none |
| `General` | extended with `&` type and operation |
| `IEEEReal` | `setRoundingMode` only accepts `TO_NEAREST` |
| `Int`=`Int32` | `~`, `*`, `+`, `-`, `div`, `quot`, `abs` don't raise `Overflow` |
| `Int8`, `Int16` | as above |
| `FixedInt`=`LargeInt`=`Int64` | as above |
| `Int`{$N$}`Vector`,`Int`{$N$}`Array` | none |
| `IO` | none |
| `List` | none |
| `ListPair` | none |
| `Math` | none |
| `Option` | none |
| `OS` | none |
| `OS.FileSys` | `tmpName` creates a file, no `access` with `A_EXEC` |
| `OS.Path` | none |
| `OS.Process` | `system` doesn't do console I/O |
| `Real`=`Real64`=`LargeReal` | no `fromDecimal`, `toDecimal` |
| `RealVector`,`RealArray` | none |
| `String`=`WideString` | none |
| `Substring`=`WideSubstring` | none |
| `TextIO` | no `getPos`*, `setPos`*, no functional I/O |
| `Time` | none |
| `Timer` | only wall-clock time is measured |
| `Vector` | none |
| `Word`=`Word32` | none |
| `Word8`, `Word16` | none |
| `LargeWord`=`Word64` | none |
| `Word`{$N$}`Vector`,`Word`{$N$}`Array` | none |

Table 2: Basis structures implemented in SML.NET

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

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