Moscow ML .Net Owner's Manual

Version 0.9.0 of November 2003

Niels Jørgen Kokholm, IT University of Copenhagen, Denmark
Peter Sestoft, Royal Veterinary and Agricultural University, Copenhagen, Denmark

This document describes Moscow ML .Net 0.9.0, a port of Moscow ML 2.00 to the .Net platform. The focus is on how Moscow ML .Net differs from Moscow ML 2.0.

Three other documents, the *Moscow ML Owner's Manual* [7], the *Moscow ML Language Overview* [5] and the *Moscow ML Library Documentation* [6] describe general aspects of the Moscow ML system.

Moscow ML implements Standard ML (SML), as defined in the 1997 *Definition of Standard ML*, including the SML Modules language and some extensions. Moreover, Moscow ML supports most required parts of the SML Basis Library. It supports separate compilation and the generation of stand-alone executables.

The reader is assumed to be familiar with the .Net platform [2].

Contents

The Moscow ML home page is `http://www.dina.kvl.dk/~sestoft/mosml.html`

1 Characteristics of Moscow ML .Net

Unlike most other ports of Moscow ML, this port is not based on porting the Caml Light runtime, but is based on the creation of a new backend that generates .Net CIL code. The system generates managed code that uses CLR types to represent ML values. The 0.9.0 version of Moscow ML .Net should be considered alpha quality.

Currently, Moscow ML .Net runs on the MS .Net Framework SDK implementation of .Net (versions 1.0 and 1.1). The intention is that it should run on the Mono [3] and dotGNU portable .Net [1] platforms when these mature.

The .Net version will typically be 2-4 times as slow as the Caml Light based classic version of Moscow ML on the same hardware. The .Net version will typically use several times more memory, with consequent implications for performance. Also, the .Net garbage collector has a somewhat harder time coping with the large number of small objects typically created by SML programs. The relative performance of the .Net version will be much worse if your ML program uses exceptions for flow control: throwing an exception in the .Net runtime is not exactly fast.

The language implemented is the same as Moscow ML 2.0 with a couple of platform-mandated variations. In particular, the meaning of the `prim_val` mechanism has been adapted to the new platform (section 3.2) and an experimental more user-friendly variant `clr_val` is provided (section 3.2.5 and 3.3.2). Some new values have been added to the built-in units, see section 1.3.

There are some minor differences in the formatting of reals. System-dependent exceptions, in particular `Syserr` and `Io`, may be formatted differently. In fact, the exact system exception raised may be different from that raised by the same ML program a Caml Light based Moscow ML port to the same platform.

Most of the Moscow ML 2.0 libraries have been ported and adapted to the .Net version, see section 1.4.

Moscow ML .Net includes ports of the `mosmllex`, `mosmldep` and `cutdeps` utilities, but not `mosmlyac`, which is written in C. However, the parser support in the .Net version is compatible with files generated by Moscow ML 2.0 versions of `mosmlyac`.

As of version 0.9.0, the on-line help system is not distributed with the installation kit.

1.1 Compiling and linking

The .Net version compiles ML signatures (in an interface file `unitid.sig` or generated from an implementation file `unitid.sml`) to .ui files just like classic Moscow ML, but the two .ui file formats are not binary compatible.

Implementation (`.sml`) files may be compiled in either netmodule, dll or exe mode. The output format may be a textual CIL assembler (`.il`) file or a .Net managed assembly. When the output format is set to managed assembly, netmodule mode is not available. The mode and output format is determined by command-line parameters (section 1.2). If you are compiling an `.sml` file in order to evaluate it (later) you should choose exe mode. If you are compiling an `.sml` file in order to load it from other ML programs, you should choose dll mode.

If you are compiling several `.sml` files and want to build a single `.exe` file you should compile all files to `.il` files, the main file in exe mode and the others in netmodule mode. Subsequently you may create a combined `.exe` file from all the `.il` files by using the MS .Net Framework `ilasm` utility. Libraries of ML structures can be built in a similar way. To access such libraries from another ML program you may use the `-netlib` command-line option or the `addLibdll` function in the interactive system when compiling the client, or set library paths by a special launcher program in a language such as C#. This is how the Moscow ML batch compiler `mosmlnetc` and the interactive system `mosmlnet` load their private library units.

1.2 Command-line options

The Moscow ML .Net batch compiler `mosmlnetc` and the interactive system `mosmlnet` accept the same command-line options as in Moscow ML 2.0, and the following additional ones:

`-target {netmodule|dll|exe}`
: (`mosmlnetc`) Determines the mode: the kind of target (file) to generate.

`-rtcg`
: (`mosmlnetc`) Use .Net runtime code generation to generate a .Net managed assembly instead of the default `.il` textual assembler. However, using the `-rtcg` flag with `-target netmodule` will create a dll file, not a netmodule file.

`-lam_debug`
: (`mosmlnetc` and `mosmlnet`) Dump the intermediate lambda code to the terminal from the compiler backend. (For debugging only; this option will go away).

`-rtver` s
: (`mosmlnetc`) Set the version string in the assembly header of the compiled file to s instead of the version string of the compiler. This option is only useful when building a new compiler and will only be honored when creating the output as textual assembler (when `-rtcg` is not specified).

`-netlib` dllfile
: (`mosmlnetc` and `mosmlnet`) Add file dllfile to the ML library search path. This permits access to user-defined libraries of ML structures. In the batch compiler `mosmlnetc` this is relevant only in exe mode. In the interactive system `mosmlnet` this initializes the `libdllPath` list (section 1.3). The dllfile may be given as a relative or absolute file name and will be converted to a full (strong) assembly name unless prefixed by `!`. This option can be given multiple times.

`-reference` dllfile
: (`mosmlnetc` and `mosmlnet`) Cause `prim_val` to search dllfile also for an fname without an explicit assembly reference. This is similar to the `/reference` option of the C# compiler. In the interactive system `mosmlnet` this initializes the `extdllPath` list (section 1.3). The dllfile may be given as a relative or absolute file name and will be converted to a full (strong) assembly name unless prefixed by `!`. This option can be given multiple times.

Notes:

- In the current version of the interactive system `mosmlnet` the `compile` family of functions always use runtime code generation and always generate dll files, corresponding to options `-rtcg` and `-dll`.
- The default output may be changed to .Net assembly (`-rtcg` and `-exe`) in the release version of Moscow ML .Net.

1.3 Additional primitives in the built-in units

There are a few extra values in the Meta unit:

`dump : unit -> unit`
: Save the dynamic assembly used for interactive evaluation to the file Topfakefakefake.exe and quit. This is for debugging only and will probably go away in the release version.

`libdllPath : unit -> string list`
: Returns the list of extra libraries of compiled ML implementation files searched by the `load` function. Initialized from the `-netlib` options (section 1.2).

`addLibdll : string -> unit`
: Adds a dll to the end of the list of libraries of compiled ML implementation files searched in addition to the standard Mosml.Library.dll by the `load` function. Name translation as for the `-netlib` option (section 1.2).

`extdllPath : unit -> string list`
: Returns the list of extra assemblies in which to search for `prim_val` fnames without explicit assembly refs. Initialized from `-reference` options (section 1.2).

`addExtdll : string -> unit`
: Adds a dll to the end of the extdllPath list. Name translation as for the `-netlib` option (section 1.2).

Notes:

- Extra Meta values are planned for controlling the target mode and file format generated by the `compile` family of functions.

1.4 The libraries

Most of the library structures of Moscow ML 2.0 have been copied to the .Net version without source code changes or with only very slight source code changes. The slightly changed structures include `Int` (32-bit precision, different `maxInt` and `minInt`), `Word` (32-bit word size), and `Weak` (different underlying implementation). Only the IO structures have had the ML source code changed considerably. The necessary porting work has mainly been done on the runtime side, which is now implemented in C#.

The following libraries from Moscow ML 2.0 are not present in Moscow ML .Net yet. The plan is to support most of these in the release version of Moscow ML .Net.

- Regex
- Socket
- Dynlib
- Gdbm
- Gdimage
- Mysql
- Polygdbm
- Postgres
- Signal
- Unix

The functionality of the old Callback structure can be achieved by different means in Moscow ML .Net, see section 3.

The integer structures Int and Word are 32 bit (compared to 31 or 63 for Caml Light based versions).

The Array/Vector structures will in a later version support lengths up to 2^{31} – currently a 2^{22} limit is still in force.

The String structures internally mostly use 16-bit characters as in .Net System.String and System.Char[], but the Moscow ML .Net implementation not entirely systematic in this respect.

The TextIO structure builds upon .Net System.IO.TextReader and System.IO.TextWriter and thus should support Unicode, but as for strings, this should be cleaned up.

2 Installation

This section describes installation of the binary distribution kit. It assumes that you have a version of MS Windows that supports the MS .Net Framework SDK 1.0 or 1.1 and that the .Net Framework SDK is already installed.

Moscow ML .Net is distributed as a zip file, which can be found at the Moscow ML homepage `http://www.dina.kvl.dk/~sestoft/mosml.html`. To install, simply download the archive and unpack somewhere on your local hard disk (Moscow ML .Net will not run from a network drive if you use the default security configuration of the .Net Framework). The batch compiler is started by the `bin\mosmlnetc.exe` managed executable, and the interactive system is started by the `bin\mosmlnet.exe` managed executable. You may add the Moscow ML .Net `bin` subdirectory to the environment variable `PATH` in your MS Windows setup.

To allow the `mosmlnetc` and `mosmlnet` executables to find the compiled interface files for the Moscow ML .Net library without having to supply the `-stdlib` option, you may create a `MOSMLNETLIB` environment variable in your MS Windows setup; its value should be the full path to the Moscow ML .Net `lib` subdirectory.

The startup time of the batch compiler and interactive system will be quite substantial (5–10 seconds) unless you install a number of assemblies into the global assembly cache and precompile them with the `ngen` utility of the MS .Net Framework SDK. This can be conveniently done by running the `lib\gaciall.bat` script included with Moscow ML .Net. You may need to be logged in as an administrator to successfully run the script, and the `ngen` and `gacutil` .Net utilities must be in a directory mentioned by the `PATH` environment variable.

Warning: the `gaciall.bat` script will remove *all* assemblies with the same names as those being installed — regardless of the version — from the global assembly cache and the `ngen` cache before installing new versions. This is harmless unless you want to run several versions of Moscow ML .Net side by side.

To uninstall Moscow ML .Net just remove the directory in which you unpacked the zip archive and remove any `Mosml.*` assemblies from the global assembly cache and `ngen` cache with the `gacutil` utility.

3 External programming interface

3.1 How external assemblies are found and loaded

Compiled ML programs (including the batch compiler and the interactive system) will need to access several kinds of .Net assemblies to run:

- The basic `Mosml.Runtime.dll` which contains runtime support methods and data, including the .Net types used to represent ML values at runtime (see section 3.2.3).
- Compiled ML implementation files.
- .Net class members referenced via the `prim_val` or `clr_val` mechanism.

First, every compiled ML program needs the assembly file `Mosml.Runtime.dll`. In will be loaded by the .Net runtime via an assembly reference in the compiled ML program, using a strong name to point to the version of Mosml.Runtime.dll used at compile time (or a version requested by a `-rtver` option to the batch compiler; section 1.2). Unless you have directed the .Net runtime to do otherwise in a configuration file, `Mosml.Runtime.dll` will only be searched for in the global assembly cache and in the directory containing the start assembly of the compiled ML program. Note that you will have to either recompile your ML programs or preserve the old `Mosml.Runtime.dll` when you upgrade Moscow ML .Net.

Secondly, the .Net code of a separately compiled ML implementation unit accessed at run time will be searched for in:

- The start assembly of the running application.
- The path given by the `string[]` field `libdlls` in class `Mosml.Runtime`, listing assembly names, either full (usually strong) assembly names with a prefixed `@` or explicit assembly file names.
- The path given by the `string[]` field `libdirs` in class `Mosml.Runtime`, listing directories to search for unit dlls by name.

In the two latter cases, the code will be loaded dynamically using one of the .Net methods `System.Reflection.Load` (for full assembly names) or `System.Reflection.LoadFrom`. A Moscow ML unit, say `unitA`, will be searched for by looking for a class `Mosml.unitA.main` in the start assembly, the assemblies of `libdlls` and finally in assemblies named `unitA.dll` in the directories of `libdirs`. For `mosmlnetc` and `mosmlnet`, the path arrays are initialized in the C# launcher programs. For an interactive session, `libdirs` is initialised from the `-I` command-line options and accessible to ML programs as `Meta.loadPath`, whereas `libdlls` is initialised from the `-netlib` command-line options and accessible to ML programs as `Meta.netlibPath`. The `Main()` entry point af an ML implementation file compiled in exe mode will initialize from the `-I` and `-netlib` command-line options given when it was compiled. Note: for loading assemblies with full assembly names, the comments in the next paragraph on usual .Net loading applies in this case too.

Thirdly, .Net class members referenced by a `prim_val` mechanism in an implementation file or an interactive declaration will be referenced in the compiled code by an ordinary .Net memberref using an assemblyref to point to the assembly containing the referenced member. Normally, a strong name reference is used. Thus the relevant assembly is loaded via the usual .Net loading: If the referenced assembly is `L.dll`, then `L.dll` must be found either in the global assembly cache or in the directory of the start assembly – unless the .Net load path for the application has been changed by a configuration file. See section 3.2 for a description of how the assemblyref is constructed.

3.2 How to call a .Net static method from Moscow ML .Net.

External (.Net) static methods, written in C# for example, can be accessed and called through the `prim_val` mechanism. For instance (from `src/mosmllib/Math.sml`):

```
prim_val sqrt : real -> real = 1 "sml_sqrt";
```

This binds variable `sqrt` to a 1-argument function of type `real -> real`, implemented by a static method

```
public static Value sml_sqrt(Value x)  ...
```

in class `Mosml.Stdlib`, whose source is found in `src/runtime/Stdlib.cs`. The type `Value` (actually `Mosml.Value` defined in `Mosml.Runtime.dll`) is the .Net base class of all ML value representations. The method referenced by `prim_val` must always have a signature of the form `(Value,...,Value)->Value` with the specified number of arguments.

There is currently no way to access instance methods or overloaded static methods via the `prim_val` mechanism.

Class Stdlib is a container for static methods to be referenced as `prim_val`s in the standard library. A `prim_val` defined with a .Net method name neither containing an assembly ref nor a namespace part will point to a method in this class.

If the method name on the `prim_val` right-hand side contains an assembly ref or a namespace part, then a static method in a class in that assembly and/or namespace will be called.

For example, this binds variable `rtcg_create_assembly` to the method `create_assembly` from class `RTCG` in namespace `Mosml`:

```
prim_val rtcg_create_assembly : string -> rtcg_t = 1 "RTCG::create_assembly";
```

Similarly, this binds variable `Mosml_load_one` to the method `load_one` from class `main` in namespace `Mosml.Top`:

```
prim_val Mosml_load_one : string -> unit = 1 "Top.main::load_one";
```

More generally, you can specify the assembly, the namespace and the class that declares the method that you need. For instance, the full specification of the `sml_sqrt` method indicated above would be:

```
prim_val sqrt : real -> real = 1 "[Mosml.Runtime]Mosml.Stdlib::sml_sqrt";
```

Here, the assembly is `Mosml.Runtime` (in file `Mosml.Runtime.dll`), the namespace is `Mosml`, the class is `Stdlib`, and `sml_sqrt` is the method.

3.2.1 An example

Assume that the C# file `Hello.cs` contains this class declaration:

```
using System;
using Mosml;
public class MyTest {
  public static Value Print(Value v) {
    System.Console.WriteLine(v);
    return Value.unit;
  }
}
```

This file contains a reference to the `Mosml` namespace in the `Mosml.Runtime` assembly. To compile it into a library `Hello.dll` use:

```
csc /target:library /reference:c:\mosmlnet\bin\Mosml.Runtime.dll Hello.cs
```

Assume the SML file `hello.sml` contains the following declarations:

```
prim_val hello : string -> unit = 1 "[Hello]MyTest::Print";

val _ = hello "Hello, world!\n";
```

Then running

```
mosmlnet hello.sml
```

in the directory holding `Hello.dll` will produce this result:

```
C:\tmp>mosmlnet hello.sml
Moscow ML .Net version 0.8.7 (October 2003)
Enter ‘quit();’ to quit.
[opening file "hello.sml"]
> val hello = fn : string -> unit
Hello, world!

[closing file "hello.sml"]
```

3.2.2 Passing arguments and using results

Arguments are passed by value (not ref or out), as objects of subclasses of class `Value` from namespace `Mosml`, declared in file `src/runtime/Values.cs`. The ML value class hierarchy is outlined in section

3.2.3. It is up to the external (non-ML) .Net static member to do the correct marshalling of values to and from the ML representation.

3.2.3 Representation of ML Values

The following is an overview of the .Net classes used to represent ML values. A more detailed description may be found in section 2 of Moscow ML .Net Internals ([4]).

```
Value                    All Moscow ML .Net values (abstract)
  MLInt                  32-bit integers (SML int and char, C# int)
  MLFloat                64-bit floating-point (SML real, C# double)
  MLAbstractString       (Internal use) (abstract)
    MLString             Strings (SML string, C# String)
    MLByteArray          (Internal use)
    MLCharArray          (Internal use)
  MLBlock0               Tuples, records, datatypes, vector
    MLBlock1             Tuples, records, datatypes, vector
      ...                ...
        MLBlock6         Tuples, records, datatypes, vector
          MLBlockInf     Tuples, records, datatypes, vector
  MLInChannel            Input channel (abstract)
    MLInFileStream       SML BinIO.in_channel
    MLTextReader         SML TextIO,BasicIO.in_channel
  MLOutChannel           Output channel (abstract)
    MLOutFileStream      SML BinIO.out_channel
    MLTextWriter         SML TextIO,BasicIO.out_channel
  MLClosure_             SML functions (abstract)
    Concrete closures
  MLWeak                 SML weak pointers
  MLWeakVector           SML weak pointer arrays
  MLDirHandle            (Internal use)
  MLExcReturn            (Internal use, Experimental)
  RTCG                   (Internal use)
  ILG                    (Internal use)
  MLClrAny               (Experimental)
```

3.2.4 Notes

In a future version, the `prim_val` mechanism will be changed so that an fname without an explicit assembly reference will be searched for in the list of dlls defined by the `-reference` command-line option and the `addExtdll` interactive function, in addition to the standard `Mosml.Runtime.dll` and `mscorlib.dll`. Perhaps the use of explicit assembly references (as in `[Hello]MY...`) in the .Net method name will then be made obsolete.

3.2.5 An experimental auto-marshalling import mechanism: `clr_val`

The `clr_val` mechanism is an experimental variant of `prim_val` that simplifies calls from Moscow ML .Net to static .Net methods.

The `clr_val` declaration, like `prim_val`, takes a (partial) name of a static .Net method, but no type specification. The supplied name is looked up via reflection at compile time and if a matching static and not overloaded .Net method is found, a corresponding Moscow ML type is constructed and attached to the declared value. If no matching method is found or the method name is overloaded, a syntax error

exception is raised by the compiler. The .Net method is looked for as a complete name in the basic `mscorlib` and `Mosml.Runtime` assemblies or in an assembly explicitly named within square brackets in the declaration.

The Moscow ML type corresponding via `clr_val` to a .Net method is constructed in the following way. The simple .Net value types (in C# notation) `int`, `uint`, `string` and `double` map to their direct SML counterparts `int`, `word`, `string` and `real`. A .Net `Array` type of one of the simple types maps to the corresponding SML `vector` type. All other .Net types map to an opaque type (`prim_type`) identified by the .Net type name. The compiler will generate glue code doing any necessary mapping between .Net values and their corresponding Moscow ML .Net internal representations. As an example, a `static double f(double a,double b)` method signature will map to the ML type `real * real -> real`.

As an example we can access the static method `Pow` in the `System.Math` namespace of the `mscorlib` assembly and call it in the following way:

```
Moscow ML .Net version 0.8.7 (October 2003)
Enter ‘quit();’ to quit.
- clr_val c = "System.Math::Pow";
> val c = fn : real * real -> real
- c(3.0,4.0);
> val it = 81.0 : real
```

If we have compiled the following `Ext.cs` program to `Ext.dll`:

```
//compile with
//  csc /target:library Ext.cs
public class Ext {
  //...
  public static string[] thearraytest(int i,double d) {
    return new string[] {(i+7).ToString(),(d+7).ToString()};
  }
  //...
}
```

we may call the `thearraytest` method from Moscow ML .Net as in

```
Moscow ML .Net version 0.8.7 (October 2003)
Enter ‘quit();’ to quit.
- clr_val b = "[Ext]Ext::thearraytest";
> val b = fn : int * real -> string vector
- b (1000,1000.8);
> val it = #["1007", "1007,8"] : string vector
```

More examples are available in the `Ext.cs` and `clr.sml` files in the `src/doc/internals` directory of the source distribution of Moscow ML .Net.

There are very few static and non-overloaded methods in the standard .Net libraries, so the current `clr_val` mechanism is mainly useful for accessing methods one develops oneself in C#. It would not be hard to extend the mechanism to work for additional simple types such as `bool`. With a little more work one could support instance methods and constructors, without which the opaque constructed types are quite useless. It would also not be hard to support overloading and explicit (checked) casting between the opaque constructed types. But the current `clr_val` mechanism would be at best only marginally more convenient than `prim_val` at accessing libraries such as `Windows.Forms` or `GTK#`, where one would still need one declaration for each method of each class one wants to call. A mechanism of auto-constructing on demand less opaque ML types in the form of ML structures corresponding to .Net types would be nice.

Note: the current implementation will identify two `prim_type`s constructed from the same .Net type

in the same Moscow ML compilation unit, but not when constructed in two different Moscow ML compilation units.

3.3 How to call an ML function from .Net

To call an ML function (or access some other kind of ML value) declared (and visible) in a compiled SML implementation file one must mimic the way another ML unit would access the value.

If the ML function to be accessed is `fun1`, declared in unit `unitA`, we must first load and evaluate `unitA` together with all its ML dependencies. Then we must find `fun1` in the export table of `unitA` and get a reference to the .Net representative of `fun1`. Finally, we must create correct ML representatives for the .Net values we wish to supply to `fun1`, call `fun1` with these arguments by the closure call conventions of Moscow ML .Net and transform the return ML value representative to the corresponding .Net standard type.

The full procedure is explained in the following example. For the full story of the closure implementation, see the accompanying document *Moscow ML .Net Internals* [4].

3.3.1 Example

Assume that we want to call the SML function `fac` defined in unit `L_ext` on the integer 7 and access the value `otte` defined in the same unit from a `C#` program, where `L_ext.sml` is

```
val otte = 3+5;
val fac = let
    fun su a 0  = a
      | su a n = su (n*a) (n-1)
in su 1 end;
```

This can be achieved as in the following `C#` program, `T_smlcall.cs`, assuming `L_ext.sml` is compiled to a dll file `L_ext.dll` in the directory `C:\tmp`:

```
using System;
using Mosml;
public class T_smlcall {
  public static void Main(string[] args) {
    string uname = "L_ext";
    //Setup lib path and load L_ext unit
    Runtime.set_libdirs("C:\\tmp","");
    Runtime.load_and_init_unit("T_smlcall.cs",uname);
    //Fetch the fac value
    Value fac = Runtime.get_unit_val(uname, "fac", Runtime.loaded_units);
    //Create an ML representative for 7
    Value seven = new MLInt(7);
    //Call fac on 7
    Value mlretval = ((MLClosure)fac).Eval1(seven);
    //Extract int result from ML representative:
    int netretval = ((MLInt)mlretval).val;
    System.Console.WriteLine("fac(7)={0}",netretval);
    //Read an int val from ML:
    Value mlotte = Runtime.get_unit_val(uname, "otte", Runtime.loaded_units);
    int netotte = ((MLInt)mlotte).val;
    System.Console.WriteLine("otte={0}",netotte);
  }
}
```

To run this example create `L_ext.sml` in `C:\tmp` and perform the following command there:

```
mosmlnetc -target dll -rtcg L_ext.sml
```

This will create `L_ext.dll` in `C:\tmp`. Then go to, say `C:\temp`, create `T_smlcall.cs` there and do

```
csc /reference:C:\mosmlnet\bin\Mosml.Runtime.dll T_smlcall.cs
```

which will create `T_smlcall.exe`. Finally run that program to produce the result:

```
C:\temp>T_smlcall
fac(7)=5040
otte=8
```

3.3.2 Experimental, easier export of ML values via `exportVal`

An experimental mechanism for exporting certain ML values from a compiled unit is provided by the `exportVal : 'a -> 'a` function in the `General` unit.

The non-function types that may be exported are the simple ML types `int`, `word`, `string` and `real` together with vectors of these (recursively). The exportable function types are simple curried types `'a -> 'b -> ...-> 'z`, where the constituent types are exportable non-function and non-record types, and where `'z` may be `unit` corresponding to a `void` method.

A warning is printed at run-time if one tries to export a non-exportable ML value.

As an example, assume that file `valexp.sml` contains

```
val k = exportVal(fn u => fn v => u+v+1);
```

and that it is compiled to `valexp.dll` using the command

```
mosmlnetc -dll -rtcg exp.sml
```

Then as for any top-level declaration `val k = ...` the value `valexp.k : int -> int -> int` will be visible from other Moscow ML compilation units, but in addition `exportVal` causes the `valexp.dll` assembly to contain a corresponding .Net method with this signature:

```
public static int Mosml.valexp.export.k(int a, int b);
```

This method may be called as shown by this program `client.cs`:

```
// Compile with
//   csc /reference:valexp.dll client.cs
public class Client {
  public static void Main() {
    System.Console.WriteLine(Mosml.valexp.export.k(3,4));
    System.Console.WriteLine(Mosml.valexp.export.k(5,4));
  }
}
```

The first call of `Mosml.valexp.export.k` will load `valexp.dll`, execute and bind its top-level declarations, evaluate `valexp.k 3 4` and then return the result. The second (and subsequent) calls will just evaluate and return the expression. The implementation of `Mosml.valexp.export.k` consists of a call to `Eval` in the closure class for `valexp.k` and glue code for marshalling and unmarshalling arguments and result.

Non-function values are exported as read-only properties.

Several examples are available in the files `valexp.sml` and `client.cs` in the `src/doc/internals` directory of the source distribution of Moscow ML .Net.

The `exportVal` function must be applied at the outermost level of a Moscow ML compilation unit on the right-hand side of a value declaration that would be visible from other ML units. If several exports of the same name is done, only the last one will take effect. It is legal to apply `exportVal` in a nested expression as in `let val u = exportVal 6 in u+7 end;` but `exportVal` is silently ignored in this case.

One may apply `exportVal` in an interactive session. In this case, the interactive system will write some diagnostic messages to the terminal, which may be useful for experimentation purposes, but since the 'exported' function/property will live in a temporary dynamic assembly one cannot actually access it from a C# program.

References

[1] Dotgnu portable.net web site. At http://www.southern-storm.com.au/portable_net.html.

[2] Microsoft .NET Framework web site. At http://msdn.microsoft.com/netframework/.

[3] Mono web site. At http://www.go-mono.org/.

[4] N. J. Kokholm and P. Sestoft. *Moscow ML .Net Internals, version 0.9.0*, November 2003.

[5] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Language Overview, version 2.00*, June 2000. Available from ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmlref.pdf.

[6] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Library Documentation, version 2.00*, June 2000. Available from ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmllib.pdf.

[7] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Owner's Manual, version 2.00*, June 2000. Available from ftp://ftp.dina.kvl.dk/pub/mosml/doc/manual.pdf.