Moscow ML .Net Internals

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 some of the internals of Moscow ML .Net 0.9.0, a port of Moscow ML 2.00 to the .Net platform. If you intend to modify Moscow ML .Net, you may find this document useful. Otherwise, read the *Moscow ML .Net User's Manual* [1] instead.

Contents

1 Structure of compiled files

1.1 Compiled interfaces

A compiled interface file has suffix `.ui` and contains an MD5 checksum followed by a serialized ML signature value. The serialization is hand-coded (this is around 10 times faster than the built-in .Net serialization) and very similar to the Moscow ML 2.0 serialization, although not binary compatible.

1.2 Compiled implementation files

A compiled implementation file generated by mosmlnetc is a .Net assembly: either an executable (.exe) or a library (.dll). It may be instructive to look at the result of disassembling such a file using ildasm while reading this section.

A compiled implementation file contains an assembly manifest that includes strongname references to mscorlib.dll and Mosml.Runtime.dll and any other assemblies referenced via the prim_val or clr_val mechanisms.

All code compiled from an implementation file, unitid.sml, will belong to the Mosml.unitid namespace. This will hold one class called main; one class for each toplevel declaration (roughly), named INIT_i where i is the declaration's sequence number; closure classes named CL_i_j where i is the enclosing declaration's sequence number and j is the closure's position in that declaration; and classes named ENV_CL_i_j for holding the free variables of closures:

```
namespace Mosml.unitid {
  public class main {...}
  public class INIT_001 {...}
  ...
  public class INIT_N {...}
  public class CL_001___1 {...}
  public class ENV_CL_001___1 {...}
  public class CL_001___2 {...}
  ...
}
```

Closure classes are described in Section 3 below. An INIT_i class will just have a single method Eval() which, when invoked, will evaluate the corresponding declaration.

The main class of a compiled unit will look as if declared like this:

```
using System.Collections;
using Mosml;
public class main {
  public static string checksum;
  public static Dependency[] dependencies;
  public static Hashtable exports;
  public static Value[] globals;
  public static Value[] imports;
  public static void init_imports(Hashtable loaded_units);
  public static void module_init(Hashtable loaded_units);
}
```

The checksum field is initialised to the corresponding .ui file checksum. The dependencies array is initialized with pairs of the form (checksumX,unitidX) for each unit that this unit loads directly. The globals field will hold global values declared as the unit is evaluated. The exports field will map exported names to indexes into globals. The imports field will cache values referenced from other units and will be filled when init_imports(...) is run. Finally, module_init(...) will run init_imports(...) and then all INIT_i.Eval() methods of this unit.

If unitid.sml was compiled in exe mode the main class will furthermore have a static Main() method that contains an .entrypoint assembly directive and when invoked will set up Runtime.libdirs, load and evaluate any ML dependencies and finally invoke module_init().

1.3 Unit loading

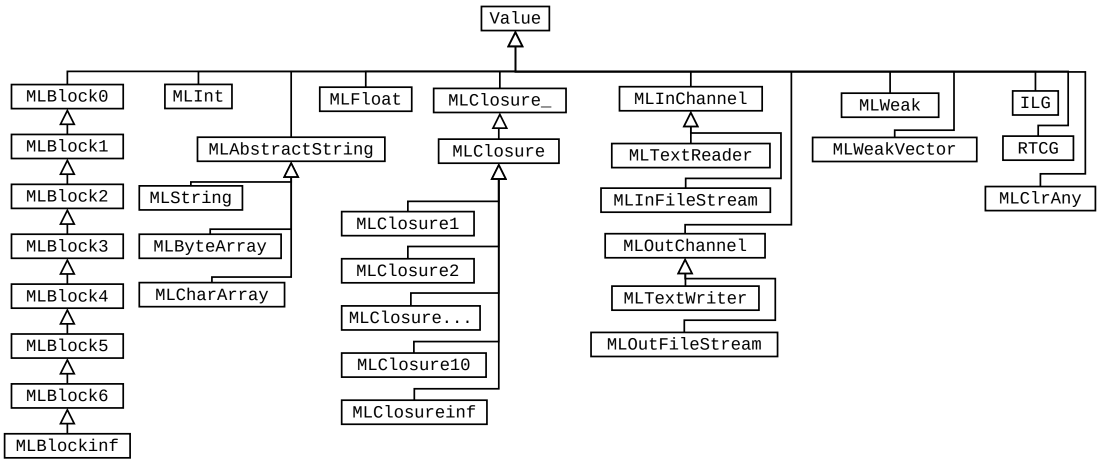
Loading of a unit will load any units on which it depends, and then invoke the unit's `module_init()`, which in turn calls the unit's `init_imports(...)` and then all its `INIT_i.Eval()` methods.

1.4 External linking of ML values

An SML function can be called from a C# program or another .Net language using the `Runtime.get_unit_val` utility function as described in [1, Section 3.3].

2 ML value representation details

All representatives of ML values belong to subclasses of the abstract class `Mosml.Value`, or just `Value`, defined in `Mosml.runtime.dll` with the source code in `src/runtime/Values.cs`:



The `Value` class itself just contains a few utility static fields and a few abstract method declarations, the most important of which is the `eqvals` method for testing equality of ML values.

2.1 Numeric types

The integral ML types `int`, `word`, `word8`, `char` are all represented by `MLInt` that simply is `Value` with a public `System.Int32` field `val` added. An ML value corresponding to a .Net `int` is created using the `MLInt(int)` constructor; the .Net `int` can be recovered again from the `val` field.

Similarly, a floating-point number (of ML type `real`) is represented as an `MLFloat` that has a `System.Double` field called `val`.

2.2 String types

An ML string is represented by a concrete subclass of `MLAbstractString`. Most strings will be represented by `MLString` that is just `Value` with a `System.String` field `val` added. Inside string libraries and IO libraries, string value representatives may be created as `MLByteArray` or `MLCharArray` objects (with mutable `val` fields of type `byte[]` respectively `char[]`) and such representatives may leak from the libraries. Normally one translates a .Net string to an ML value representative using the `MLString(string)` constructor and translates the other way by the `ToString()` method valid on all of `MLAbstractString`.

The situation is currently a little messy. The strings should really be considered arrays of 8-bit characters, but use the 16-bit .Net string and char types to a large extent for representation.

2.3 Block types

ML tuples, records, datatypes, refs, vectors and arrays are all represented as "blocks", which should be thought of as `Value[]`s with an associated 8-bit tag field. To save almost half the object creations and deletions by block operations, the most obvious implementation of blocks has been modified to the following chinese box implementation (simplified view):

```
public class MLBlock0 : Value {
  public byte tag;
  public int len;
  public Value this[int i];
}
public class MLBlock1 : MLBlock0 {
  public Value v0;
}
public class MLBlock2 : MLBlock1 {
  public Value v1;
}
...
public class MLBlock6 : MLBlock5 {
  public Value v5;
}
public class MLBlockinf : MLBlock6 {
  public Value[] vinf;
}
```

We need the deep hierarchy because the compiler back end does not know when accessing, say field number 3 (i.e. `v2`) of a block, which precise type the given block is, just that it has the specific field `v2`, and thus must be represented by an `MLBlock3` or a subclass.

Tuples are represented directly with the tuple elements as block fields in order and tag 0. Records are represented as tuples in alphabetic order of the labels. Datatype values are represented as blocks with the tag being the rank in alphabetic order of the constructor name (starting with 0) and the fields being the elements of the toplevel tuple in the constructor argument. References are represented as blocks with tag 250 and the referenced element as single field. Vectors are represented just like tuples and arrays are vector refs.

The easiest way to access the tag is to access the `tag` field. The easiest way to access individual fields is to use the `this[int i]` method via array syntax in C#. There are suitable constructors for all the block types.

2.4 Closure types

SML function values (closures) are represented by subclasses of the abstract `MLClosureXXX` classes, with the `XXX` corresponding to the number of arguments in the original `fun` declaration on which the closure is based. (In Moscow ML curried function declarations are, for efficiency purposes, kept curried and not translated to non-curried ones before compilation). The `MLClosureXXX` classes have the relevant number of Value fields for saving arguments supplied in undersaturated calls. Values saved from the function declaration environment will be saved in fields allocated in the concrete subclasses of the MLClosureXXX.

The MLClosure class defines the common interface of all closure representatives:

```
Value Eval1(Value arg1);
Value Eval2(Value arg1, Value arg2);
...
Value Eval10(Value arg1, ..., Value arg10);
Value Evalinf(Value[] args);
```

An SML function application with M arguments will be compiled to pushing the representatives of the arguments on the CLR stack and then calling EvalM on the closure representative if $\mathtt{M} \leq \mathtt{10}$ and else collecting the representatives of the arguments in a Value[] array and then calling Evalinf on the closure representative. The number 10 is chosen as a number large enough that the fallback case for large argument counts should not occur in normal SML code.

MLClosure and the MLClosureXXX have to use the CIL directive “tail.” to make the CLR use tail calls for efficiency. Therefore these classes cannot be written in (current versions of) C#. Instead, they are generated as textual CIL assembler by the src/genclosure/genclosure.sml program of the source distribution. For pedagogical reasons only, the source distribution contains mockup C# source code correspoding to part of the generated CIL code; this mockup is in src/doc/internals/closure_mockup.cs. The MLClosure_ class in the figure above is a trivial intermediate class defined in src/runtime/Values.cs just for defining an abstract Eval1 method for the benefit of a piece of code in src/runtime/Stdlib.cs that calls back to ML from C# using this interface. The MLClosure_ enables the src/runtime/*.cs files to be compiled independently of the detailed CIL implementation of MLClosure.

See Section 3 for more details on the closure implementation.

2.5 Other types

The types MLInChannel, MLOutChannel, MLInFileStream, MLTextReader, MLOutFileStream and MLTextWriter represent IO channels as used in the BasicIO, TextIO, BinIO and NonstdIO library units. The first two mentioned classes are abstract utility classes, while the other four are simple wrappers of their CLR counterparts.

The SML types Weak.weak and Weak.array are represented by MLWeak and MLWeakVector. They are simple wrappers of a System.WeakReference respectively a System.WeakReference[] array.

MLDirHandle is used for FileSys.dirstream SML values. The CLR does not seem to have any analogue of the usual C opendir etc. interface, and so MLDirHandle actually caches the whole directory listing in an internal array on creation.

RTCG and ILG are used for runtime code generation, inside the compiler/interactive system. The RTCG type is a conglomerate of an assemblybuilder, a modulebuilder together with typebuilders and methodbuilders as required. The ILG type wraps an IL generator for a specific method.

MLExcReturn has been used for experimental work with exception handling see Section 4.

MLClrAny is used for representing values of the special prim_types created by clr_val, see [1]. It contains an object field for the CLR object and a string field for the (full) CLR type name corresponding to the prim_type of the SML value represented.

3 Closure implementation details

3.1 Function representation

Moscow ML .Net functions are represented by objects of subclasses of the abstract classes `Mosml.MLClosureN`, `N=1,...` defined in `Mosml.Runtime.dll` based on source code in the files `src/runtime/Values.cs` and `src/runtime/MLClosure.il` as noted above.

The `Mosml.MLClosureN` classes are themselves subclasses of `Mosml.MLClosure`.

The function `f` defined by

```
fun f x_1 x_2 ... x_N  = ...
```

will be represented by an object `rep_f` of a subclass, `Rep_f`, of `Mosml.MLClosureN`. Class `Rep_f` will implement a method `EvalN(Value A_1,...,Value A_N)` with a body containing the compiled body of the definition of `f` (with the arguments in reverse order, see below). The real name of `Rep_f` will be something like `Mosml.<unitid>.CL_<i>_<j>`, where `i` is the index in the implementaion file `unitid.sml` of the toplevel declaration containing the definition of `f` and `j` is the index of the definition of `f` inside said declaration. A function defined by a partial application of `f` to `n(<N)` arguments will be represented by an object of class `Mosml.<unitid>.CL_<i>_<j>` too, but with the arguments supplied so far saved in fields `Mosml.MLClosureN::savarg1` to `Mosml.MLClosureN::savarg<n>`.

The description in the preceding paragraph is not completely correct if `N` is greater than the value `Cil_glob.large_arg_count` (currently 10). In that case `Mosml.<unitid>.CL_<i>_<j>` will be a sublass of a class `Mosml.MLClosureinf`. The following discussion will ignore function definitions and applications with more than `Cil_glob.large_arg_count` arguments. The reader is referred to the source of `Mosml.Runtime.dll` for information on this case.

3.2 Function application

A function application

```
(g e_1 e_2 ... e_N)
```

will be compiled the following virtual instance call on `rep_g`:

```
Mosml.MLClosure::EvalN(rep_e_N,...,rep_e_2,rep_e_1)
```

The reason for the reverse order is that the compiler back-end will evaluate and push the argument expressions to the stack in increasing order of the index, which in .Net corresponds to the call shown. This is fine if `G` were defined by a

```
fun g x_1 x_2 ... x_N = ...
```

of the same arity since then the compiled body of `G` will be called directly. Else a system of dispatcher methods in the `Mosml.MLClosureN` classes comes into play.

3.3 The dispatcher methods

The `Mosml.MLClosure` superclass of all the `Mosml.MLClosureN` classes has abstract methods `EvalM(Value A_1,...,Value A_M)` for `M` from 1 to `Cil_glob.large_arg_count`. These methods are implemented in all `Mosml.MLClosureN` classes as dispatcher methods that work as follows:

1. If the total count of arguments (including already saved ones) is less than `N`, the newly supplied arguments are saved in `savarg<_>` fields following any earlier saved arguments in a clone of `"this"` and the clone is returned.
2. If the total count of arguments equals `N`, the relevant `Mosml.<unitid>.CL_<i>_<j>::EvalN` compiled function body is called by a virtual tail call to `Mosml.MLClosureN::EvalN` with the earlier saved arguments and the newly supplied ones as arguments.
3. If the total count of old and new arguments exceeds `N`, say by `M`, the relevant compiled function definition body, `Mosml.<unitid>.CL_<i>_<j>::EvalN`, is called by a virtual (non-tail) call to `Mosml.MLClosureN::EvalN` using the saved and some of the newly supplied arguments. The return Value must be an `MLClosure` and is called with the remaining `M` of the newly supplied arguments by a virtual tail call to `Mosml.MLClosure::EvalM`.

This scheme, as described, has the following weakness. In the following situation:

```
fun f x_1 x_2 ... x_N = ...
val g = f e_1 e_2 ... e_M = ...
(g e'_1 e'_2 ... e'_N)
```

where `M<N`, the code generated by the compiler for the application of `g` will result in erroneously dispatching `Mosml.<unitid>.CL_<i>_<j>::EvalN` with the arguments of the application of `g` instead of correctly dispatching the method `Mosml.MLClosureN::EvalN` in the base class of `Rep_g` (which is also the base class of `Rep_f`) with these arguments.

The solution chosen is to have a check whether `savarg1 != null` in the beginning of the body of `Mosml.<unitid>.CL_<i>_<j>::EvalN` and in that case make an immediate non-virtual tail call with the supplied arguments to the dispatcher `Mosml.MLClosureN::EvalN`. For that to work, the (first) calls in steps 2. and 3. above must actually start by cloning `"this"` and set `savarg1=null` in the clone before calling (virtually) `Mosml.MLClosureN::EvalN` on the clone.

The weakness and consequent insertion of guard code in every function body implementation could be avoided by letting the method implementing the function definition body be called something like `Mosml.<unitid>.CL_<i>_<j>::realEval` and letting the first call in steps 2. and 3. be a virtual call to `Mosml.MLClosure::realEval`. This would then have the effect that every function application would be made as 2 CLR method calls. We have chosen to optimize the case of direct manifest calls.

3.4 Tail calls

On the CLR, a method call will be performed as a tail call if prefixed by the `tail.` directive and immediately followed by a `ret` instruction (if allowed by code access security). Tail calls on CLR, unlike on typical runtime systems specially made for functional languages, is not faster than ordinary method calls, but take roughly the same amount of time. The main advantage of tail calls on CLR is the less stack used for deeply recursive calls, and perhaps an accompanying better locality of stack references.

We try to make SML tail calls be compiled to CLR tal calls in the following way. The compiler backend is written in CPS, where the code is generated backwards, so that we always can see the code which will use the result of the currently emitted code. When we are going to emit a method call, we see if the next instruction will be a `ret` and if so insert a `tail.` prefix of the `call` or `callvirt` instruction. To recognize all/more? tail call opportunities, `ret` instructions are propagated backwards in the code stream through branch instruction as much as possible. Special care has been made to allow tail calls from within an exception handler, the difficulties being that one cannot leave a CLR catch block by a `ret`, see Section 4.

Since tail calls in the CLR are much slower than branches, we compile a saturated tail call to a function within its own body to a branch instead. For such self-tail-calls, the compiler will emit code to place the arguments in the slots ot the stack allocated for the original call to this closure body (instead of pushing the arguments) and then branch to a label at the start of the compiled closure body. Currently, only let bound functions with at most 10 curried arguments will have self-tail-calls detected and optimized in this way. In particular, a fun defined at the outermost evaluation level will not have self-tail-calls optimized.

3.5 Large argument counts

The discussion above ignored the case where a (curried) function definition has a very large (>10) argument count, and the case where a function value is applied to a very large number of arguments.

A closure with more than 10 args in its definition will be a subclass of `MLClosureInf`. This is completely analogous to the lower `MLClosureXXX` classes, but the dispatcher methods will save the arguments not in individual `Value` fields, but in a `Value[] arginf` field of `MLClosureInf` until full and then call `realEval()` virtually on the closure class. The concrete implementation of that method will contain the real compiled closure body expecting to find its arguments in the `arginf` array field (not on the stack).

When Applying a closure value to more than 10 args, the compiled code will call the concrete `MLClosure.Eval(Value[])` method on an array of the args. If the number of arguments is `10K+M`, with integers `K>0` and `0<M`$\leq$`10`, the body of `MLClosure.Eval(Value[])` will split up the argument array into `K` pieces of size `10` and one piece of size `M`, call `MLClosure.Eval10` virtually `K` times on the given closure respectively intermediate results with arguments from the pieces of size `10`, and finally tail call `MLClosure.EvalM` virtually on the remaining `M` arguments from the last piece.

In the combined case, one could copy arguments directly from the argument of `MLClosure.Eval(Value[])` to the `Value[] arginf` field of `MLClosureInf`, but since these cases are not supposed to appear in practical code, we have ignored this optimization opportunity.

4 Exception handling

4.1 ML exceptions are CLR exceptions

Moscow ML .Net uses CLR exception handling to implement ML exception handling. ML exception values are represented as refs to certain tuples. Moscow ML .Net defines a CLR class, `Mosml.MLException`, a subclass of `System.Exception` (but not of `Mosml.Value`), which wraps an ML exception value inside a CLR exception.

The CLR exception handling architecture imposes some restrictions:

- The CLR evaluation stack must be empty on entry to a try block and on exit from a try or catch block.
- One can only exit a try or catch block with the `leave` (for label) instruction or by throwing an exception.
- It is expensive to throw an exception in CLR.

The first issue is no problem for a language where the exception handling is based on statements, but it is not hard to violate in SML although rare in practical code. The solution chosen for Moscow ML .Net is to save the ML evaluation stack in freshly allocated local variables before entering the try block and restore the stack afterwards – see a little more details below.

The second issue in particular implies that one cannot leave a catch block with a `ret` instruction, which means that we cannot propagate `ret` through branches to a position inside the catch block to recognize tail call opportunities there. The solution is to have the compiled ML exception handling code execute outside the catch block and just have a small stub execute inside.

The catch clause of a compiled handle expression always catches `System.Exception`. The stub code inside the catch block first calls a utility filter method on the caught exception. If the filter recognizes the

CLR exception as either a wrapped ML exception value or a CLR exception that has a corresponding ML equivalent (like arithmetic overflow), the filter returns the ML exception value, else the original exception is `rethrown` and will eventually be caught by the outermost exception handler of the application. The stub code then stores the ML exception value in a local variable, from where the code outside will load it, and finally leaves the catch block. Right after the catch block comes the compiled ML handler code.

The following pseudo code illustrates the compiled code for the ML expression `e1 handle handlematch`:

```
save ML evaluation stack into CLR local variables (if necessary)
try {
 compiled e1
 store top of CLR stack in local variable SCRATCH_VALUE
 leave LABEL1
}
catch (Exception e) {
 translate e to ML exception or rethrow it
 store top of CLR stack (the ML exception value) in local variable V_1
 leave LABEL2
}
LABEL2:
load local variable V_1
compiled handlematch
store top of CLR stack into local variable SCRATCH_VALUE
LABEL1:
restore ML evaluation stack (if relevant)
load local variable SCRATCH_VALUE
```

To raise an exception from inside a handled expression in ML, with the notation in the pseudo code above, the ML exception value is stored in the appropriate local variable, `V_1` and then a `leave LABEL2` instruction is executed. This avoids the overhead of throwing, catching and filtering a `Mosml.MLException`. Raising an exception outside a handler will result in code that wraps the ML exception value inside an `Mosml.MLException`, which is subsequently `thrown`.

4.2 An alternative implementation of ML exceptions

To try to avoid the high overhead of throwing a CLR exception when raising an ML exception outside of handled expressions, an alternative has been tried for code in a function body. Instead of throwing an exception, one empties the evaluation stack except for the ML exception value, wraps that in a special `Mosml.MLExcReturn` object and executes `ret`. Then every compiled function application must be immediately followed by code that checks whether the return value is an `Mosml.MLExcReturn` and in that case raises the exception: If inside a try block by unwrapping the ML exception value from the return value and leaving; if not in a try block but still in a function body by once again returning the special value; and if neither in a try block nor in a function body, by rewrapping the ML exception in a `Mosml.MLException` and throwing it. This experimental implementation greatly improved the running time of certain examples that use exceptions heavily for flow control. However, the idea was abandoned because of its negative impact on the majority of programs, which do not use exceptions for non-exceptional control flow.

References

[1] N. J. Kokholm and P. Sestoft. *Moscow ML .Net User's manual, version 0.9.0*, November 2003.