Interlanguage Working Without Tears: Blending SML with Java

Nick Benton Andrew Kennedy

Microsoft Research Ltd., Cambridge, U.K.
{nick,akenn}@microsoft.com

Abstract

A good foreign-language interface is crucial for the success of any modern programming language implementation. Although all serious compilers for functional languages have some facility for interlanguage working, these are often limited and awkward to use.

This article describes the features for bidirectional interlanguage working with Java that are built into the latest version of the MLj compiler. Because the MLj foreign interface is to another high-level typed language which shares a garbage collector with compiled ML code, and because we are willing to extend the ML language, we are able to provide unusually powerful, safe and easy to use interlanguage working features. Indeed, rather then being a traditional foreign *interface*, our language extensions are more a partial *integration* of Java features into SML.

We describe this integration of Standard ML and Java, first informally with example program fragments, and then formally in the notation used by The Definition of Standard ML.

1 Introduction

Functional language implementations nearly all provide some way to call external C functions [4, 9, 12], but direct interworking with a low-level, non-typesafe, language with no garbage collection is never going to be easy or pretty. Most functional programmers never use the foreign interface except via functionally-wrapped libraries written by compiler experts.

The importance of good foreign language interfaces for functional languages is now widely recognised, particularly given the wider trend towards mixed-language component-based programming. Recent years have seen a number of functional interfaces to language-independent component architectures, notably COM (see, for example, [6, 11]) and CORBA [10]. To quote [6]:

> "Programming languages that do not supply a foreign-language interface die a slow, lingering death – good languages die more slowly than bad ones, but they all die in the end."

To appear in the 4th ACM International Conference on Functional Programming, September 1999, Paris, France.

Download the current version of the MLj compiler from `http://www.dcs.ed.ac.uk/home/mlj/`

One of the main motivations for the design of MLj [1], a compiler for Standard ML that generates Java bytecodes, was to provide a foreign-language interface that would give ML programmers access to the extensive libraries available in Java. We expected to be able to implement a much more useful and convenient foreign language interface to Java than is possible for C because the 'semantic gap' between ML and Java is comparatively small:

- Both languages are strongly typed, and there are some good correspondences between the basic types in the two languages. For example:
  - The numeric types are a close match;
  - Strings are immutable vectors of (possibly-null) characters in both languages;
  - Array bounds are checked in both languages;
  - Neither language has explicit pointer types.
- Both languages have automatic storage management. Furthermore, because we compile ML code to Java bytecodes, we actually use the same garbage collector for ML and Java objects so there is no need to deal with references between independently managed heaps.
- Exception handling in the two languages is similar.

There are, however, still significant differences between the two languages; the concepts of objects, classes, inheritance and dynamic method dispatch which are at the heart of Java have no natural counterparts in ML, and Java lacks many features of ML, such as parametric polymorphism and higher-order functions. We did not wish to burden users with the complexities of a separate Interface Definition Language (IDL), as is used for COM and CORBA, so the natural (if unusually bold) approach was to extend SML with types and terms corresponding to Java constructs. Doing this well is a very tricky language design problem: the (unattainable) ideal would be an object-oriented conservative extension of ML which still maintains the spirit of ML but which also corresponds naturally and predictably to the type system of Java.

The first version of MLj (release 0.1) extended SML with types and syntax which covered essentially all of Java. However, the two worlds were kept fairly separate. For example, the type `Java.int` represented Java language integers (passed to or from external Java code) and converting between this type and ML's `int` type required an explicit

```
open javax.swing java.awt java.awt.event
_classtype SampleApplet () : JApplet ()
with local
  val prefix = "Counter: "
  val count = ref 0
  val label = JLabel(prefix ^ "0", JLabel.CENTER)
  fun makeButton (title, increment) =
  let
    val button = JButton (title:string)
    val listener =
      ActionListener ()
      with
        actionPerformed(e : ActionEvent option) =
        (count := !count + increment;
         label.#setText(prefix ^ Int.toString(!count)))
      end
  in
    button.#addActionListener(listener);
    button
  end
in
  init () =
  let
    val SOME pane = this.#getContentPane ()
    val button1 = makeButton ("Add One", 1)
    val button2 = makeButton ("Add Two", 2)
  in
    pane.#add(button1, BorderLayout.WEST);
    pane.#add(label, BorderLayout.CENTER);
    pane.#add(button2, BorderLayout.EAST);
  end
end
```

Figure 1: A sample applet in MLj

`Java.fromInt` or `Java.toInt` coercion, which would actually disappear during compilation because ML's `int` is represented as Java's `int` in the compiled code. The extensions for creating classes from within ML included new keywords for all Java class and method attributes, even where these overlapped conceptually with existing ML concepts. For example, the namespace management of Java's static final fields and static methods provided by classes overlaps the scoping of ML value bindings provided by the module system, and mutable fields are similar to ML `ref`s.

Experience with the MLj 0.1 extensions showed that they were extremely useful, but that there was plenty of scope for improvement. Amongst other things, the strict separation of primitive types was unnecessarily annoying, there was too much baroque syntax associated with the embedding of Java in ML (some of which was rarely used because there were better ways of using existing ML constructs to achieve the same effect) and one often wished to treat Java fields and methods in a more first-class way.

This article describes the revised design of the inter-language working extensions that is being implemented for the next version of MLj and the rationale for that design. The new extensions are much more of an *integration* of Java concepts within ML than an *interface* between the two languages, though for compatability with SML, we have stopped short of trying to design a fully-fledged 'natural' object-oriented extension of ML. We also sketch a formalisation in the style of the Definition of Standard ML [13].

We assume a knowledge of ML and a passing familiarity with Java. Note that for clarity of presentation we omit certain features of the ML-Java interface, in particular, the use and definition of Java interfaces, and the formalisation of certain aspects.

2 Design goals

We wanted ML code to be able to read and write external Java fields, call external Java methods and treat external Java objects in a first-class way – storing them in ML data structures, passing and returning them from ML functions. We also wanted ML code to be able to define new Java classes which extend existing external classes (and implement external interfaces) with methods written in ML. These classes should be accessible and usable either from within the ML program or by other external Java code.

In addition to these basic goals, we wanted the interface to have a certain 'flavour'. At first we took the view that the interface could be slightly ugly when compared with ML itself, as it would only be used by library-writers to provide a functional wrapper around an existing Java package; indeed, the strangeness of any new syntax would serve as reminder that the programmer was doing something reserved for experts. However, once we discovered how useful and pleasant it was to have straightforward, safe interoperability with Java's standard library code and third-party Java applications (and sometimes even to develop new mixed-language applications from scratch), we began to think that this should be made as convenient and natural as possible. Therefore in the most recent version of the foreign-language interface we have aimed to provide:

- *Simplicity*: the syntax used to access Java classes and to create new ones should be as lightweight as possible so that a programmer can use it without to much of a mental context-switch. To attain simplicity as many ML concepts as possible were re-used (for example, packages are identified with structures and sub-packages with substructures) but this was done only where it makes sense semantically (for example, static methods are like ML functions but non-static (virtual) methods are not).
- *Compatibility*: correct programs written using SML'97 should typecheck and execute without alteration. This severely constrained the syntax extensions that we could use, but we believe that our design is tasteful and unobtrusive. We have not quite managed to preserve all the equations which hold in SML'97 since, for example

  `("h"^"i","h"^"i")` and
  `let val x="h"^"i" in (x,x) end`

  are not contextually equivalent (as Java can check object identity).
- *Safety*: one pleasant aspect of Java is that type safety is built-in. In contrast to C, programs cannot corrupt the store, leave pointers dangling, and so on. However, ML goes further, insisting that values are bound explicitly at their definition, whereas in Java the possiblity of a default `null` value can lead to a `NullPointerException`. To retain the spirit of ML, we are quite strict about values of Java class and array types, insisting that `null` values are checked for explicitly.

Java notion	*ML notion/new syntax*
primitive type	base type
class name	type identifier
array type	array type
`null` value	`NONE`
`void` type	`unit` type
multiple arguments	single tuple argument
mutable fields	`ref` type
package	structure
subpackage	substructure
importing a package	opening a structure
static field	value binding
static method	function binding
non-static field access	`.#`
non-static method invocation	`.#` and `.##`
object creation	function binding
casts	`:>` in expressions
`instanceof`	`:>` in patterns
class definition	`_classtype`
private fields	local declarations
private/package access methods	signature matching

Table 1: Analogies between ML and Java

- *Power*: Many of the constructs that we provide correspond quite closely with those found in the Java language, both syntactically and semantically, but there were a few places where we were able easily to improve on Java, for example making methods and fields first-class, and by introducing a type-case construct.

Just as important as our goals is one of our non-goals. We decided that, at least at this stage, we would not allow arbitrary ML values (such as closures or values of user-defined datatypes) to be passed to external Java code. This (a) seemed less useful than the ability to pass values the other way (any external Java code that could do anything interesting with an ML value would be better written in MLj), (b) would potentially compromise safety by, for example, allowing Java code to mutate supposedly immutable ML values, and (c) would require us to use a predictable uniform representation for ML values, which would inhibit many of the optimisations performed by our compiler. However, it seemed unduly restrictive to prevent classes created from within an MLj program but which do *not* get exported to the external Java world from making free use of values of arbitrary ML types. This introduces a slight complication into our model, as some MLj classes are now regarded as exportable and others as purely internal – the details are explained later.

Table 1 summarizes the correspondence between Java concepts and existing SML concepts or new features that we introduced. The code in Figure 1 illustrates many of these, and we will use it as a running example throughout. Two buttons control the incrementing of a common counter that is displayed in the centre. The code is for JavaSoft's *Swing* GUI framework.

3 Types

3.1 Primitive and class types

The Definition of Standard ML [13] does not specify the base types of the language, and the Standard Basis library specification does not prescribe particular sizes for numeric types, instead leaving it up to the implemention. Hence we were able to match ML base types to Java primitive types, avoiding the need for unpleasant coercions when passing values of base type to and from Java.[1] Table 2 gives the correspondence that we used.

Java type	ML type
`boolean`	`bool`
`byte`	`Int8.int`
`char`	`char`
`double`	`real`
`float`	`Real32.real`
`int`	`int`
`long`	`Int64.int`
`short`	`Int16.int`
`java.lang.String`	`string`
`java.lang.Exception`	`exn`
`java.math.BigInteger`	`IntInf.int`
`java.util.Calendar`	`Date.date`

Table 2: Correspondence between types in Java and ML

The first eight entries in the table are the Java primitive types. The remainder are class types, which can be referred to from within ML using the same syntax as in the Java language, so for example `java.lang.StringBuffer` is a Java string buffer and `java.awt.Color` is a Java colour. This syntax works because of the interpretation of Java packages as structures and subpackages as substructures, discussed later.

There arises the question of which Java types should be given equality status within ML, that is, permitted as arguments to ML's polymorphic equality operator `=`. For the base types listed in Table 2, the Basis Library forces the issue: all are equality types except for `real`, `Real32.real`, `exn` and `Date.date`. For other class types, there are three alternatives: equality by identity (Java's `==` operator), user-defined equality (Java's `equals` method), or no equality at all. The first can be rejected as being outside the spirit of ML, and would in any case conflict with equality-by-value on strings and big integers; the second also does not have the right flavour, as `=` is an equivalence relation for all applicable types in ML, and user-definability would break this; therefore, we decided to exclude general class types from having equality status.

3.2 Arrays

Java arrays have virtually identical semantics to ML arrays: their size is fixed at creation-time, indexing starts at zero, equality is based on *identity* not value, and an exception is raised upon out-of-bounds access or update. Therefore the ML array type constructor `array` corresponds to Java's array type constructor `[]`.

The single glitch is Java's unsound covariant subtyping on arrays, and its corresponding dynamic check on array update to fix up the unsoundness. For ML arrays implemented using Java arrays, this check always succeeds and is therefore unnecessary, but unfortunately must introduce some performance overhead.

[1] In fact, we depart slightly from the basis specification in adopting Java's use of Unicode for characters and strings; the basis prescribes ASCII 8-bit characters.

3.3 Null values

In the Java language, variables with class or array types (known collectively in the Java literature 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 `NullPointerException` if their main operand is `null`.

ML 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 in Java. We wished to retain this safety in our Java extensions to ML, and so interpret a value of Java reference type as "non-null instance".

Nevertheless, when a Java field of reference type is accessed from ML or a value of reference type is returned from an external Java method invoked by ML, it may have the value `null` and this must be dealt with by the ML code. Also, it should be possible to pass null values to Java methods and to update Java fields with the null value. Fortunately the ML 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 Java reference type that cross the border between ML and Java as values of an `option` type. For example, a stand-alone Java application must have a method `main` with the following prototype:

```
public static void main(java.lang.String []);
```

Inside ML, the single argument to this method is treated as a value of type

```
string option array option
```

meaning "a possibly-null array containing possibly-null strings".

3.4 Field types

Java fields qualified by the keyword `final` are immutable and their types are interpreted as indicated above, using `option` to denote the possibility of null values for objects or arrays. For example, the field declared in the `java.lang.System` class as

```
public static final java.io.InputStream in;
```

is interpreted as having type `java.io.InputStream option`.

Fields not qualified by `final` are mutable and their types are interpreted using ML's `ref` type constructor. So a field declared by

```
public static byte[] b;
```

is given the ML type `Int8.int array option ref`.

3.5 Method types

Java 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, Java has a syntax for multiple arguments but ML does not, so methods with multiple arguments are given a single tuple argument type. (The alternative, a curried function type, presents no problems but the syntax of method invocation inside ML would then be very different from the syntax in Java). Finally, when arguments and results are objects or arrays, their types are interpreted using the `option` type constructor as described earlier.

Consider the following two prototypes taken from the system class `java.lang.String`:

```
public static java.lang.String
  copyValueOf(char[], int, int);
public java.lang.String toString();
```

Their types are interpreted respectively as

```
char array option * int * int -> string option
```

and `unit -> string option`.

3.6 Overloading and implicit coercions

Java 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, method invocations implicitly coerce arguments up through the class hierarchy. The combination of these features can lead to ambiguity, which Java compilers resolve 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.

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

We do not allow ambiguity to be resolved by Java-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 Accessing Java from ML

4.1 Packages, subpackages, and classes

If one ignores the class hierarchy and non-static fields and methods (i.e. a non-object-oriented fragment of Java), then Java packages and classes can be seen (and are used) as a minimal module system, providing a way of carving up the namespace for fields and methods into manageable chunks. We therefore chose to model them using the SML module system.

Top-level packages in the Java world are reflected in ML as a collection of top-level structures, with subpackages as substructures. Classes are reflected as three separate bindings: as type identifiers, as values of function type used to construct instances of the class (discussed later), and as structures containing value bindings that reflect static fields and methods. For example, within the package `java.lang` (reflected as a structure `lang` inside a top-level structure java), the class Integer is mapped to an ML type identifier Integer, to a value identifier Integer, and to a structure Integer. There is no problem having types, values and structures sharing a name as they inhabit different namespaces in SML.

4.2 Import as open

open *package*
open *class-name*

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

Opening of packages-as-structures is analogous to Java's import *package*.* construct; for example, the declaration open javax.swing in Figure 1 is roughly equivalent to import javax.swing.*. However, when used with classes-as-structures the open mechanism is more powerful, permitting unqualified access to static fields and methods. Also, subpackages become visible as structures: the sample program opens java.awt and then uses event.ActionEvent to refer to the java.awt.event.ActionEvent class.

4.3 Fields

class-name . *field-name*
exp .# *field-name*

Static fields are mapped to ML value bindings. Fields qualified by final really are treated as simple values; an example of this is the BorderLayout.WEST constant used in Figure 1. Non-final fields are interpreted as ML value bindings with ref types. The implementation permits these to be used in a first-class way, improving on Java. To make this possible, values of Java reference type are compiled as objects with 'reader' and 'writer' methods; immediate assignment or dereferencing compiles to code as efficient as that produced by a Java compiler. There is however a small performance hit for ordinary ML ref values that have Java types, as if it wasn't for Java these could be implemented more efficiently by performing access and update inline.

As mentioned earlier, we provide explicit provision for null values through the use of option types. For example, the colour constants provided in the java.awt.Color class have Java declarations such as

```
public static final java.awt.Color pink;
```

and might be accessed in ML by

```
val SOME pink = java.awt.Color.pink
```

Non-static fields (instance variables) are accessed by the new *exp*.#*field-name* syntax. (It is not possible to use a simple dot notation because there would be no means of distinguishing such expressions from those used for static field access). Here *exp* is an ML expression of class type and *field-name* is a Java field name. As with static fields, non-final fields can be used as first-class ref values.

4.4 Methods

class-name . *method-name*
exp .# *method-name*

Static methods are mapped to ML value bindings of function type. Again, we improve on Java, and permit such functions to be used in a first-class way, by eta-expanding where necessary:

```
val colours =
  map (valOf o java.awt.Color.getColor)
    ["red", "green", "blue"]
```

Here we have made use of the automatic insertion of the SOME coercion as discussed in Section 3.6, as the getColor method is interpreted as having the type

```
string option -> java.awt.Color option
```

but values of type string are passed to it.

Non-static (virtual) method invocation uses the syntax *exp*.#*method-name*, where *exp* is an expression of class type and *method-name* is a method defined or inherited by that class. There are many examples of this in Figure 1: the label.#setText invocation again illustrates a coercion from string to SOME string, and the pane.#add invocation illustrates class coercions (to Component option) and overloading (as the add method has many alternative argument types).

4.5 Object creation

class-name exp

In Java, new instances of a class are created using the syntax new *class-name*($arg_1, \ldots, arg_n$), where arg_i are the arguments to one of the constructors defined by the class.

We avoid the need for any new syntax in MLj by binding the class name itself to the constructor function. If there is more than one constructor, then the binding is overloaded. For example, the constructors for javax.swing.JButton appear as bindings to the identifier JButton inside the structure javax.swing. This is illustrated in Figure 1 in the construction of JLabel and JButton objects.

As with methods, constructors can be used as first-class values, and implicit coercions are applied using the same rules. For example:

```
val labels = map javax.swing.JLabel ["A", "B"]
```

4.6 Casts and typecase

(expression)	*exp* :> *ty*
(pattern)	*id* :> *ty*

A new syntax is introduced (borrowed from O'Caml [14, 12]) to denote Java-style casts. It can be used to cast an object up to a superclass:

```
val c = (JButton "My button") :> Component
```

Explicit coercions are sometimes required when passing Java objects to ML functions, as coercions are only applied implicitly when invoking Java methods.

The same syntax can also be used to cast an object down to a subclass, with Java's ClassCastException thrown if the actual class of the object is not compatible. A safer alternative that combines downcasting with Java's instanceof is the use of :> inside ML patterns. This can be used to provide a construct similar to the TYPECASE of Modula-3 [3] and other languages. Suppose that a parser was written in Java and used subclassing of a class Expr to represent different node types. Then we could traverse the parse tree using case analysis:

```
case (expr : Expr) of
  ce :> CondExpr => ...code for conditionals...
| ae :> AssignExpr => ...code for assignment...
```

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*.

The new construct is a pattern like any other. It can be used in `val` bindings, such as

```
val x :> java.awt.Window = y
```

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

```
val result = (f y)
  handle e :> java.lang.SecurityException => 0
```

in order to handle (and possibly deconstruct) Java exceptions. The *order* in which handlers appear is important. In the example below, `IllegalArgExn` subclasses `RuntimeException` so if the handlers were switched the second handler would never be reached.

```
fun test x = (do_some_java x)
  handle y :> IllegalArgExn => f y
       | _ :> RuntimeException => g x
```

Finally, the behaviour of Java's `e instanceof c` can be emulated by `case e of _ :> c => true | _ => false`.

5 Creating Java classes in ML

So far we have seen how to access external Java code from ML. We now turn to the problem of creating new Java classes inside ML.

5.1 Static classes

As we have observed already, static fields and methods are orthogonal to the object-oriented nature of Java, and are reflected as bindings in SML structures. We follow this correspondence in allowing the *export* of SML structures as Java classes containing only static members. This requires no new language constructs – instead, a compiler directive is used to specify which top-level structures are to be exported as named classes.

The signature of the structure is interpreted in the following way:

- Value bindings with function types are exported as public static methods with the same name, provided that the function type is exportable.
- Other bindings are exported as static final fields with the same name, provided that the value type is exportable.

In essence, an *exportable type* is one that safely captures the way in which a field or method can be used from the Java world. Thus, pure ML types (such as `int list`) are not permitted, as Java programs have no way of knowing how these are represented. For Java reference types, the `option` type constructor must be applied whenever it is possible for Java to construct null values. This is true for method arguments (because a Java program could pass in `null`) but not for method results or fields.

We do not provide for the export of mutable fields. Whilst Java's mutable fields can be modelled using ML's first-class `ref`s, the converse is not true, as the following example demonstrates:

```
structure S =
struct
  val x = ref 5
  val y = x
end
```

It is not possible to express this kind of aliasing using Java mutable fields. Methodologically, the absence of static non-final fields is no great loss, as it is poor object-oriented style to provide direct read-write access to what is essentially a global variable.

5.2 Creating instantiable classes from ML

`_classtype` ⟨*cmod*⟩ *class-name pat* ⟨`:` *ty exp*⟩
`with` ⟨`local` *dec* `in`⟩ *method-dec* `end`

The export of structures as classes provides a means for Java to call ML, but it does not allow for the creation of class libraries with an object-oriented interface, neither does it allow for the specialisation of existing Java classes with new instance methods coded in ML. For this we introduce a new construct whose syntax is shown above. This introduces a new class type *class-name* defined by the following elements:

- The optional class modifier in *cmod* can be `abstract` or `final` and has the same meaning as in Java.
- 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, an idea that is borrowed from O'Caml [14, 12].

  Unlike Java, multiple constructors are *not* supported; a future enhancement might allow additional constructors to be expressed as invocations of a 'principal' constructor.
- The optional *ty exp* specifies a superclass type *ty* and an argument (or tuple of arguments) *exp* to pass to the superclass constructor.
- *dec* is a set of SML declarations that are local to a single instance of the class.
- *method-dec* is set of instance method declarations, defined using the syntax already used for ordinary functions, but with optional qualifiers `abstract`, `final` and `protected` preceding the method identifiers.
- We follow Java in allowing several classes to be defined simultaneously by mutual recursion, using the keyword `and` to separate the declarations.

In keeping with tradition, and to demonstrate that classes are usable in MLj without reference to Java, Figure 2 presents a variation on the classic coloured-point example.

A striking aspect of the new construct is 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

```
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 : java.awt.Color
    and move (xinc, yinc) = this.##move (xinc*2, yinc*2)
  end
end
```

Figure 2: Coloured points in MLj

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. In order to implement this method using Java's class mechanism, the compiler will probably store `c` in a Java field, but note that the only means of accessing it from MLj is through the method provided.

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 Java, private fields for other instances *can* be accessed directly, for example, to implement an `equals` method. In MLj, this can be emulated by providing appropriate 'get' and 'set' methods for the fields, then hiding these by a signature as explained below.

The special identifier `this` has the same meaning as in Java, 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. However, we do not support `super` as its semantics in Java confusingly differs depending on whether it is used to access a field (in which case it has the same meaning as a cast up to the superclass and so is superfluous) or to invoke a method (where it has a different run-time semantics, namely to ignore the over-riding of the method in the subclass). 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 over-riding 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.

We allow references to `this` in the superclass constructor arguments and in the local declarations. Unfortunately, this opens up a type loophole as methods invoked on `this` may make use of `local` identifiers not yet bound. Outlawing `this` completely is too strong a restriction (as many classes set up initial state through methods in the superclass), and the weaker restriction of allowing only methods in the superclass to be invoked does not fix the situation as the example in Figure 3 demonstrates. The behaviour of this program is ill-defined; in fact, it is likely that the exception `List.Empty` will be raised when `m2` attempts to take the head of a list that has yet to be defined. The root of the problem is the combination of object initialisation and dynamic method dispatch on the object being initialised. The same problem exists in Java [8, §12.5], and enforcing strong restrictions would have reduced expressivity without completely closing the hole because of virtual method invocations inside an external superclass constructor.

```
_classtype C ()
with
      m () = this.#m2 ()
  and m2 () = 0
end

_classtype D () : C ()
with local
  val x = [this.#m ()]
in
  m2 () = hd x
end

val dobj = D ()
```

Figure 3: A small type loophole

As mentioned in Section 3.6, Java allows overloading of methods. We support this in `_classtype` declarations in order to extend existing Java 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
```

5.3 Class types in signatures

> `_classtype` ⟨*cmod*⟩ *class-name* ty_1 ⟨`:` ty_2⟩
> `with` *method-spec* `end`

Corresponding to the class type *declaration* there is a class type *specification* construct for SML signatures. Here ty_1 is the type of the constructor argument(s) bound by the pattern expression *pat* in the corresponding class type declaration, and ty_2 is the superclass. The method specifications *method-spec* list function types for each method.

The types in the signature must correspond exactly to those in the corresponding declaration, but methods can be omitted in the same way as value bindings can be omitted from an ordinary SML signature. This lets the programmer hide methods from users of a class (corresponding to private methods in Java), or to share methods amongst a number of classes in a single module but to hide them from clients of the module (corresponding roughly to package access in Java).

```
signature POINTSIG =
sig
  _classtype Point (int*int)
  with
        getX : unit -> int
    and getY : unit -> int
  end
  _classtype ColouredPoint (int*int*java.awt.Color) : Point
  with
    getColour : unit -> java.awt.Color
  end
end
```

Figure 4: A signature for coloured points

A similar treatment of privacy is used in Moby, another ML-style language with OO features [7].

Figure 4 presents a signature that might be used to constrain `Point` to be an 'immovable point' when used by clients of the module. In the specification for `ColouredPoint`, the methods inherited from `Point` are not listed explicitly, but are still accessible. It is not possible to over-ride or inherit a method and at the same time reduce access to it, in keeping with Java's own rules.

5.4 Inner classes

There are no restrictions on the scope in which a `_classtype` declaration can appear. As with Java 1.1, classes can be nested inside functions, or even inside methods defined in other class declarations. Variables 'captured' by such declarations are implemented using the same mechanism as used for inner classes in Java – the compiler generates instance variables that are filled in when objects are constructed.

Java also extends its `new` construct for object creation to allow the definition of a new unnamed class and at the same time create an instance. This provides a kind of first-class function mechanism and is used extensively for 'callbacks' in GUI programming. MLj supports a similar syntax:

class-name exp `with` *method-dec* `end`

It is used in Figure 1 to create an `ActionListener` object in which the `actionPerformed` method is over-ridden to provide functionality specific to a button component. (In fact, `ActionListener` is an *interface*; for conciseness we have omitted discussion of interfaces in this article).

5.5 Class types and functors

The Java language and the JVM do not currently support parametric polymorphism. Therefore we restrict the types of methods in classes to be monomorphic. However, by using SML's powerful functor construct it is possible to parameterise classes on types and values. Figure 5 gives an example. When applied to a particular type `T`, the functor provides a new class type `J` and functions `wrap` and `unwrap` that convert values between `T` and `J`. The specification of `J` in the signature of the result hides both the class constructor and its method, and thus is exportable in the sense described in the previous section. The class types `IntListWrapper.J` and `IntFunWrapper.J` can then be used in Java code to pass around objects that wrap up ML values of type `int list` and `int->int`. (If one wished to use these wrapper classes to, for example, store ML values in Java collections, one would also have to include a hash function in the class.)

```
functor Wrapper(type T) :>
sig
  _classtype J
  val wrap : T -> J
  val unwrap : J -> T
end =
struct
  _classtype J(x : T)
  with
    get() = x
  end
  fun wrap   (x : T) = J(x)
  fun unwrap (j : J) = j.#get()
end

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

Figure 5: Using functors

5.6 Exporting classes

To make a class visible to the Java world, it is *exported* using a compiler directive. Its methods must be exportable according to the same rules that were described in Section 5.1.

Non-exported classes are used only within an MLj program, so no restrictions are placed on the types of their methods. Note, however, that when a class overrides a method from a superclass its types must match exactly; a method in a non-exported class that over-rides an external Java class (or an exported MLj class) must therefore also have exportable type.

There is another restriction on what may be exported, caused by the fact that exporting classes and overriding imported methods both fix the actual class and/or method names used in the generated bytecode, which the compiler is otherwise free to choose. An ML class type may be bound to multiple type identifiers, for example via structure rebinding. However, two ML class types with the same stamp (generated when the class is defined or a functor is applied) may not *both* be exported. Together with the requirement that the superclass of an exported class must be external or exported, we believe that this allows the compiler to pick method names so as to avoid the accidental or unsound overrides which might otherwise happen when a superclass method was hidden by a signature and a subclass then 'overrode' that method. This potential conflict between object extension and width subtyping is well-known; see [15] for example.

6 Formalisation

A complete formalisation of the ML-Java interface would end up specifying the static and dynamic semantics for a substantial part of the Java language. We do not attempt to do this (the interested reader should consult [5]). Rather, we give only the *static semantics* (the typing rules) for our language extensions; moreover, we omit certain details such as access control and checking of class and method qualifiers.

6.1 Types and translations

We start by extending ML types with a new category of *class types*, ranged over by c and specified formally in the

$$\text{PrimType} = \left\{ \begin{array}{l} \texttt{bool}, \texttt{int}, \texttt{char}, \texttt{real}, \texttt{Real32.real}, \\ \texttt{Int8.int}, \texttt{Int16.int}, \texttt{Int64.int} \end{array} \right\}$$

$$\frac{\tau \in \text{PrimType} \cup \text{ClassType}}{\tau \in \text{JavaType}} \qquad \frac{c \in \text{ClassType}}{c\ \texttt{option} \in \text{JavaType}}$$

$$\frac{\tau \in \text{JavaType}}{\tau\ \texttt{array} \in \text{JavaType}} \qquad \frac{\tau \in \text{JavaType}}{\tau\ \texttt{array option} \in \text{JavaType}}$$

$$\text{JavaType} \subseteq \text{Type}$$

Figure 6: Java types

$$\begin{array}{lcl}
\mathsf{ml}(\texttt{bool}) & = & \texttt{bool} \\
\mathsf{ml}(\texttt{byte}) & = & \texttt{Int8.int} \\
\mathsf{ml}(\texttt{char}) & = & \texttt{char} \\
\mathsf{ml}(\texttt{double}) & = & \texttt{real} \\
\mathsf{ml}(\texttt{float}) & = & \texttt{Real32.real} \\
\mathsf{ml}(\texttt{int}) & = & \texttt{int} \\
\mathsf{ml}(\texttt{long}) & = & \texttt{Int64.int} \\
\mathsf{ml}(\texttt{short}) & = & \texttt{Int16.int} \\
\mathsf{ml}(c) & = & c\ \texttt{option} \\
\mathsf{ml}(T\ \texttt{[]}) & = & \mathsf{ml}(T)\ \texttt{array option} \\
 & & \\
\mathsf{ml}(T\ \texttt{(}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{)}) & = & \mathsf{ml}(T_1) \times \cdots \times \mathsf{ml}(T_n) \to \mathsf{ml}(T) \\
\mathsf{ml}(\texttt{void}\ \texttt{(}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{)}) & = & \mathsf{ml}(T_1) \times \cdots \times \mathsf{ml}(T_n) \to \texttt{unit}
\end{array}$$

Figure 7: Translation from Java types

style used by The Definition [13, §4.2]:

$$\begin{array}{lcl}
c & \in & \text{ClassType} = \text{ExtClass} \cup \text{MLClass} \\
\tau & \in & \text{Type} = \text{TyVar} \cup \text{RowType} \cup \text{FunType} \\
 & & \cup \text{ConsType} \cup \text{ClassType}
\end{array}$$

Class types are either *external* (obtained from existing Java code and represented by the fully-qualified name of the class) or are *internal* (introduced through `_classtype`):

$$ic \in \text{MLClass} = \text{TyName} \times \text{ClassType} \times \text{TypeSet} \times \text{MethEnv}$$

We write $\mathsf{class}(t, c, \vec{\tau}, ME)$ for elements of MLClass. Here t is a *stamp* that identifies the class and allows recursive reference in its definition, c is the superclass, $\vec{\tau}$ is a set of constructor types (with zero or one elements), and ME specifies the methods as a set of bindings of the form $(id : \tau)$. The set of classes that are *exported* by means of a compiler directive is given by ExpClass $\subseteq$ MLClass. We let ec range over ExpClass $\cup$ ExtClass. Finally we define JavaType $\subseteq$ Type by the inductive definition presented in Figure 6.

Figure 7 defines a total function ml that maps syntactic Java types onto their ML interpretation. It is extended to a function that maps Java method prototypes to ML function types.

The mapping from ML types to Java types is given in Figure 8. It is necessarily *partial* and splits into two variants: j^+ is used to translate types for values that are *exported* (results of functions and final fields) and j^- for values that are

$$\begin{array}{lcl}
\mathsf{j}^{\pm}(\texttt{boolean}) & = & \texttt{boolean} \\
\mathsf{j}^{\pm}(\texttt{Int8.int}) & = & \texttt{byte} \\
\mathsf{j}^{\pm}(\texttt{char}) & = & \texttt{char} \\
\mathsf{j}^{\pm}(\texttt{real}) & = & \texttt{double} \\
\mathsf{j}^{\pm}(\texttt{Real32.real}) & = & \texttt{float} \\
\mathsf{j}^{\pm}(\texttt{int}) & = & \texttt{int} \\
\mathsf{j}^{\pm}(\texttt{Int64.int}) & = & \texttt{long} \\
\mathsf{j}^{\pm}(\texttt{Int16.int}) & = & \texttt{short} \\
 & & \\
\mathsf{j}^{+}(ec) & = & ec \\
\mathsf{j}^{+}(ec\ \texttt{option}) & = & ec \\
\mathsf{j}^{+}(\tau\ \texttt{array}) & = & \mathsf{j}^{+}(\tau)\ \texttt{[]} \\
\mathsf{j}^{+}(\tau\ \texttt{array option}) & = & \mathsf{j}^{+}(\tau)\ \texttt{[]} \\
 & & \\
\mathsf{j}^{-}(ec\ \texttt{option}) & = & ec \\
\mathsf{j}^{-}(\tau\ \texttt{array option}) & = & \mathsf{j}^{-}(\tau)\ \texttt{[]} \\
 & & \\
\mathsf{j}^{\pm}_{\to}(\tau_1 \times \cdots \times \tau_n \to \texttt{unit}) & = & \texttt{void}\ \texttt{(}\mathsf{j}^{-}(\tau_1)\texttt{,}\ldots\texttt{,}\mathsf{j}^{-}(\tau_n)\texttt{)} \\
\mathsf{j}^{\pm}_{\to}(\tau_1 \times \cdots \times \tau_n \to \tau) & = & \mathsf{j}^{+}(\tau)\ \texttt{(}\mathsf{j}^{-}(\tau_1)\texttt{,}\ldots\texttt{,}\mathsf{j}^{-}(\tau_n)\texttt{)}
\end{array}$$

Figure 8: Translation from ML types

imported (arguments to functions). (The notation $\mathsf{j}^{\pm}$ just indicates simultaneous definition of both maps at primitive type, where they coincide). The difference between the two is that the `option` tag on a non-primitive type is *permitted* for export but *required* for import. Java does not make the distinction between non-null and possibly-null values so we have to assume the worst when importing values from Java. Observe that $\mathsf{ml} \circ \mathsf{j}^+$ and $\mathsf{ml} \circ \mathsf{j}^-$ are the identity on Java types.

The two maps are used to define a mapping $\mathsf{j}^{\pm}_{\to}$ from function types to Java method prototypes. Note that only one version of this map is required as functions can only be exported; however if Java supported first-class methods then it would make sense to define its dual $\mathsf{j}^{\mp}_{\to}$ that reversed the polarities for argument and result types.

With these definitions we can formalise the *exportable* ML types as $dom(\mathsf{j}^+)$ (for fields) and $dom\mathsf{j}^{\pm}_{\to}$ (for methods).

6.2 Relations

For two Java types τ and τ' we write $\tau \leqslant_w \tau'$ whenever values of type τ can be converted by a *widening reference conversion* [8, §5.1.4] to type τ', or by the identity, or by the injection represented by `SOME : 'a -> 'a option`. This relation is defined inductively in Figure 9. Likewise we write $\tau \geqslant_n \tau'$ whenever values of type τ can be converted by a *narrowing reference conversion* [8, §5.1.5] to type τ' or by the identity, as defined in Figure 10.

We write $\tau \leqslant_a \tau'$ whenever argument values of type τ can be converted by *method invocation conversion* to type τ', which we define to mean either widening reference conversion on Java type or the identity. Finally, for method types of the form $\tau_1 \times \cdots \times \tau_n \to \tau$ we write $\tau \leqslant_m \tau'$ whenever method type τ can be converted to method type τ'. Following Java, we support contravariance in the argument types but no variance in the result. These last two relations are given in Figure 11.

$$\frac{}{\tau \leqslant_w \tau} \qquad \frac{\tau \leqslant_w \tau'' \qquad \tau'' \leqslant_w \tau'}{\tau \leqslant_w \tau'} \qquad \frac{}{c \leqslant_w \mathsf{super}(c)}$$

$$\frac{}{\tau\ \texttt{array} \leqslant_w \texttt{java.lang.Object}} \qquad \frac{}{\tau \leqslant_w \tau\ \texttt{option}}$$

$$\frac{\tau \leqslant_w \tau'}{\tau\ \texttt{array} \leqslant_w \tau'\ \texttt{array}} \qquad \frac{\tau \leqslant_w \tau'}{\tau\ \texttt{option} \leqslant_w \tau'\ \texttt{option}}$$

$$\leqslant_w \subseteq \text{JavaType} \times \text{JavaType}$$

Figure 9: Reference widening for Java types

$$\frac{}{\tau \geqslant_n \tau} \qquad \frac{\tau \geqslant_n \tau'' \qquad \tau'' \geqslant_n \tau'}{\tau \geqslant_n \tau'} \qquad \frac{}{\mathsf{super}(c) \geqslant_n c}$$

$$\frac{}{\texttt{java.lang.Object} \geqslant_n \tau\ \texttt{array}}$$

$$\frac{\tau \geqslant_n \tau'}{\tau\ \texttt{array} \geqslant_n \tau'\ \texttt{array}} \qquad \frac{\tau \geqslant_n \tau'}{\tau\ \texttt{option} \geqslant_n \tau'\ \texttt{option}}$$

$$\geqslant_n \subseteq \text{JavaType} \times \text{JavaType}$$

Figure 10: Reference narrowing for Java types

6.3 Class lookup

We define various helper functions on classes. Their formalisation is straightforward but tedious, so for conciseness we omit it.

- $\mathsf{super}(c)$ gives the superclass of c, if it exists.
- $\mathsf{staticfields}(c)$ and $\mathsf{fields}(c)$ give the static fields and non-static fields of c as a set of elements each of the form $id : \tau$.
- $\mathsf{staticmethods}(c)$ and $\mathsf{methods}(c)$ give the static methods and non-static methods of c as a set of elements each of the form $id : \tau$ for function type τ. Constructors are treated as static methods with the name `<init>`.

Note that for fields and methods (but not for constructors) all inherited members are included. For external classes it is assumed that there is a 'pervasive' environment from which the information can be gathered. For internal classes the type c itself includes sufficient information, though it should be noted that occurrences of the type name $t \in \text{TyName}$ in types of methods and constructors must be expanded to the class definition c itself.

6.4 Classes as structures

To formalise the use of qualified identifiers for static fields and methods, we extend the range of a value environment to include a *static member reference*, written $\mathsf{member}(c, id)$, where c is a class and id the name of a field or method. Constructors for class c are represented by $\mathsf{member}(c, \texttt{<init>})$.

The initial environment E under which a program is typed is extended to include packages and classes as structures, using the member references to fill in variable environments with appropriate bindings for fields, methods and constructors.

$$\frac{\tau \leqslant_w \tau' \text{ or } \tau = \tau'}{\tau \leqslant_a \tau'} \qquad \frac{\tau_1' \leqslant_a \tau_1 \qquad \cdots \qquad \tau_n' \leqslant_a \tau_n}{\tau_1 \times \cdots \times \tau_n \to \tau \leqslant_m \tau_1' \times \cdots \times \tau_n' \to \tau'}$$

Figure 11: Method type conversion

6.5 Typing rules for language extensions

The typing rules are presented in Figure 12 in the style of the Definition. First note that TE ranges over type environments that map type identifiers to types (paired with datatype constructor environments), VE ranges over value environments, E ranges over environments that include type and value environments, and C ranges over contexts that include environments amongst other information.

Rules *statfld* and *fld* assign types to field access expressions. Rules *statmth* and *mth* do the same for methods and incorporate subsumption on method types. Rule *supmth* is similar to *mth* but starts the search at the superclass. Rule *cast* allows an expression to be cast up or down according to the widening and narrowing relations defined earlier.

Rule *patcast* deals with cast patterns. The first premise simply ensures that constructors are not rebound. As with other pattern expressions in ML, the pattern elaborates to a value environment that gives types to variables bound to the pattern (in this case, the type specified in the cast), and a type for the match itself (in this case, a type from which a value is cast).

Rule *class-dec* elaborates a class type declaration to produce a type environment TE with the new type and a value environment VE with the constructor. The first premise deals with the formal arguments to the constructor, with TE present in the context so that type constraints can refer to the new class type c. The second premise elaborates the superclass type c'. The next two premises elaborate the arguments to the superclass constructor *exp* and local declarations *dec* respectively, both typed in a context C' containing value bindings for the constructor arguments (VE_p), the constructor for the new class (VE), and `this`. Finally, the method declarations *method-dec* are elaborated in the presence of the environment E built up by the local declarations, to produce a method environment ME.

In the corresponding rule *class-spec* for signatures, notice that the constructor type ty_1 is optional, allowing for the hiding of constructors in signature matching.

7 Conclusions and further work

One immediate area for further work is improving the type inference process for our extended language. The main problem with is in resolving method overloading (the typing rules above do not specify just how and where the resolution is done). We do not wish to insist on explicit type constraints

Expressions

$$
(\mathit{statfld})\ \frac{\begin{array}{l} C(\mathit{longvid}) = \mathsf{member}(c, \mathit{id}) \\ (\mathit{id} : \tau) \in \mathsf{staticfields}(c) \end{array}}{C \vdash \mathit{longvid} \Rightarrow \tau}
\qquad
(\mathit{fld})\ \frac{\begin{array}{l} C \vdash \mathit{exp} \Rightarrow c \\ (\mathit{id} : \tau) \in \mathsf{fields}(c) \end{array}}{C \vdash \mathit{exp}\,\texttt{.\#}\mathit{id} \Rightarrow \tau}
$$

$$
(\mathit{statmth})\ \frac{\begin{array}{l} C(\mathit{longvid}) = \mathsf{member}(c, \mathit{id}) \\ (\mathit{id} : \tau) \in \mathsf{staticmethods}(c) \end{array} \qquad \tau \leqslant_m \tau'}{C \vdash \mathit{longvid} \Rightarrow \tau'}
\qquad
(\mathit{mth})\ \frac{\begin{array}{l} C \vdash \mathit{exp} \Rightarrow c \\ (\mathit{id} : \tau) \in \mathsf{methods}(c) \end{array} \qquad \tau \leqslant_m \tau'}{C \vdash \mathit{exp}\,\texttt{.\#}\mathit{id} \Rightarrow \tau'}
$$

$$
(\mathit{supmth})\ \frac{\begin{array}{l} C \vdash \mathit{exp} \Rightarrow c \\ (\mathit{id} : \tau) \in \mathsf{methods}(\mathsf{super}(c)) \end{array} \qquad \tau \leqslant_m \tau'}{C \vdash \mathit{exp}\,\texttt{.\#\#}\mathit{id} \Rightarrow \tau'}
\qquad
(\mathit{cast})\ \frac{C \vdash \mathit{exp} \Rightarrow \tau \qquad C \vdash \mathit{ty} \Rightarrow \tau' \qquad \tau \leqslant_w \tau' \ \mathit{or}\ \tau \geqslant_n \tau'}{C \vdash \mathit{exp}\ \texttt{:>}\ \mathit{ty}\ \Rightarrow \tau'}
$$

Patterns

$$
(\mathit{patcast})\ \frac{\mathit{vid} \notin \mathrm{Dom}(C) \text{ or } \mathit{is} \text{ of } C(\mathit{vid}) = \texttt{v} \qquad C \vdash \mathit{ty} \Rightarrow \tau \qquad \tau \leqslant_w \tau' \ \mathit{or}\ \tau \geqslant_n \tau'}{C \vdash \mathit{vid}\,\texttt{:>}\,\mathit{ty} \Rightarrow (\{\mathit{vid} \mapsto (\tau, \texttt{v})\}, \tau')}
$$

Declarations

$$
(\mathit{class\text{-}dec})\ \frac{\begin{array}{l} C + \mathit{TE} \vdash \mathit{pat} \Rightarrow (\mathit{VE}_p, \tau) \\ C \vdash \mathit{ty} \Rightarrow c' \\ C' \vdash \mathit{exp} \Rightarrow \tau' \\ C' \vdash \mathit{dec} \Rightarrow E \\ C' + E \vdash \mathit{method\text{-}dec} \Rightarrow \mathit{ME} \\ \tau'' \leqslant_m \tau' \to c' \end{array}}{C \vdash \texttt{\_classtype}\ \ \mathit{id}\ \mathit{pat}\ \texttt{:}\ \mathit{ty}\ \mathit{exp}\ \texttt{with local}\ \ \mathit{dec}\ \texttt{in}\ \mathit{method\text{-}dec}\ \texttt{end} \Rightarrow (\mathit{TE}, \mathit{VE}) \text{ in Env}}
$$

$$
\text{where}\quad \begin{array}{l} c = \mathsf{class}(t, c', \{\tau \to t\}, \mathit{ME}) \text{ for fresh } t \\ \mathit{TE} = \{\mathit{id} \mapsto (c, \{\})\} \\ \mathit{VE} = \{\mathit{id} \mapsto \mathsf{member}(c, \texttt{<init>})\} \\ (\texttt{<init>} : \tau'') \in \mathsf{staticmethods}(c') \\ C' = C + \mathit{TE} + \mathit{VE} + \mathit{VE}_p + \{\texttt{this} \mapsto (c, \texttt{v})\} \end{array}
$$

$$
(\mathit{method\text{-}dec})\ \frac{C \vdash \mathit{exp} \Rightarrow \tau \qquad \langle C \vdash \mathit{method\text{-}dec} \Rightarrow \mathit{ME}\,\rangle}{C \vdash \mathit{id}\,\texttt{=}\,\mathit{exp}\ \langle \texttt{and}\ \mathit{method\text{-}dec} \rangle \Rightarrow (\mathit{id} : \tau)\ \langle + \mathit{ME} \rangle}
$$

Specifications

$$
(\mathit{class\text{-}spec})\ \frac{\begin{array}{l} \langle C + \mathit{TE} \vdash \mathit{ty}_1 \Rightarrow \tau \rangle \\ C \vdash \mathit{ty}_2 \Rightarrow c' \\ C + \mathit{TE} \vdash \mathit{method\text{-}spec} \Rightarrow \mathit{ME} \end{array}}{C \vdash \texttt{\_classtype}\ \ \mathit{id}\ \langle \mathit{ty}_1 \rangle\ \texttt{:}\ \mathit{ty}_2\ \ \texttt{with}\ \ \mathit{method\text{-}spec}\ \texttt{end} \Rightarrow (\mathit{TE}, \mathit{VE}) \text{ in Env}}
$$

$$
\text{where}\quad \begin{array}{l} c = \mathsf{class}(t, c', \{\langle \tau \to t \rangle\}, \mathit{ME}) \text{ for fresh } t \\ \mathit{TE} = \{\mathit{id} \mapsto (c, \{\})\} \\ \mathit{VE} = \{\mathit{id} \mapsto \mathsf{member}(c, \texttt{<init>})\} \end{array}
$$

$$
(\mathit{method\text{-}spec})\ \frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad \langle C \vdash \mathit{method\text{-}spec} \Rightarrow \mathit{ME}\,\rangle}{C \vdash \mathit{id}\ \texttt{:}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{method\text{-}spec} \rangle \Rightarrow (\mathit{id} : \tau)\ \langle + \mathit{ME} \rangle}
$$

Figure 12: The typing rules

being added all over the place, but it does not seem possible to use the usual syntax-directed inference system with constraints solved on the fly by unification, because ambiguities are sometimes only resolvable by considering a whole compilation unit. The right thing to do is to gather up the constraints generated by the above rules and try to solve them all together for each top-level structure, and this is what we plan to do. Our current working version, however, uses essentially the same algorithm as we use for SML; this sometimes requires type constraints to be added in unpredictable (algorithm-dependent) places.

More speculative further work might include adding parameterized classes in the style of GJ [2] or defining uniform representations of ML types, to be enforced only on the ML/Java interface, so that ML values could be passed to Java code for purposes such as serialization for persistence or mobility. It might also be interesting to take the object-oriented aspects of MLj further, for example by allowing more general subtyping and overloading on ML values. Such a language would no longer be SML, however, and would probably be less natural as an object-oriented version of ML than, say, OCaml [14] because of the need to match Java. Indeed, our extensions have already become more like an object-oriented extension of SML than we originally intended.

The main limitation of our approach is that the extensions are non-standard and specific to Java. Foreign interfaces based on COM or CORBA, by contrast, allow interworking with components written in many different languages. We did try using our extensions plus a Java ORB to make ML interface to CORBA components; this was successful, but the CORBA bindings for Java were a great deal more unpleasant to use than a direct mapping of CORBA to ML would be.

Nevertheless, the interlanguage working extensions described here are (we modestly believe!) far more pleasant to use than any comparable system. There is no need to worry about linkers, interface definition languages, stubs, marshalling and unmarshalling or memory management – working with Java from ML is much like working with Java from Java, and sometimes better. In particular, our decision to map Java constructs to ML ones where possible, but not to be afraid to extend ML where such a mapping is not natural seems to be have been a good one.

The latest version of the MLj compiler is available from `http://www.dcs.ed.ac.uk/home/mlj`.

Acknowledgements

The evolution of the ML-Java interface has benefited greatly from our many discussions with George Russell, Dave Halls, Audrey Tan, Ian Stark, Bent Thomsen and Stephen Gilmore, amongst others.

References

[1] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, October 1998.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, DEC Systems Research Center, November 1989.

[4] H. Davis, P. Parquier, and N. Séniak. Sweet Harmony: the Talk/C++ Connection. In *Conference Record of the 1994 ACM Conference on Lisp and Functional Programming*, pages 121–127, Orlando, Florida, USA, 1994.

[5] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[6] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[7] K. Fisher and J. Reppy. The design of a class mechanism for MOBY. In *ACM SIGPLAN Conference on Programming Language Design and Implenentation*, May 1999.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical report, AT&T Bell Laboratories, January 1996. See `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

[10] D. Jeffery, T. Dowd, and Z. Somogyi. MCORBA: A CORBA binding for Mercury. In *Practical Aspects of Declarative Languages: First International Workshop*, volume 1551 of *Lecture Notes in Computer Science*. Springer Verlag, January 1999.

[11] X. Leroy. Camlidl user's manual version 1.0, March 1999. See `http://caml.inria.fr/camlidl/htmlman/`.

[12] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. Objective Caml user's manual, 1998. See `http://pauillac.inria.fr/ocaml/`.

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

[14] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[15] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999. To appear.