Chapter 1

O'Camelot: Adding Objects to a Resource Aware Functional Language

Nicholas Wolverson

Abstract: We outline an object-oriented extension to Camelot, a functional language in the ML family designed for resource aware computation. Camelot is compiled for the Java Virtual Machine, and our extension allows Camelot programs to interact easily with the Java object system. [1]

1.1 INTRODUCTION

The Mobile Resource Guarantees (MRG) project aims to equip mobile bytecode programs with guarantees that their usage of certain computational resources (time, heap or stack space, ...) does not exceed some agreed limit, using a Proof Carrying Code framework. Programs written in the functional language Camelot will be compiled into bytecode for the Java Virtual Machine. The resulting class files will be packaged with a proof of the desired property and transmitted across the network to a code consumer—perhaps a mobile phone, or PDA. The recipient can then use the proof to verify the given property of the program before execution. There is thus an unforgeable guarantee that the program will not exceed the stated bounds.

The core Camelot language, as described in [7], enables the programmer to write a program with a predictable (and in time provable) resource usage. However, only primitive interaction with the outside world is possible, through command line arguments and printed output. To be able to write a full interface for a game or utility to be run on a mobile device, Camelot programs must be able

[1]This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

to interface with external Java libraries. Similarly, the programmer may wish to utilise device-specific libraries, or even Java's extensive class library.

Here we describe an Object-Oriented extension to Camelot primarily intended to allow Camelot programs to access Java libraries. It would also be possible to write resource-certified libraries in Camelot for consumption by standard Java programs, or indeed use the object system for OO programming for its own sake, but giving Camelot programs access to the outside world is the main objective.

1.2 CAMELOT

Camelot is an ML-like language with additional features to enable close control of heap usage. Certain restrictions are made in order to enable a compilation process which is transparent in terms of resource usage, and to allow analysis of resource usage by various novel type systems.

One can define datatypes in the usual fashion:

```
  type 'a lst = Nil | Cons of 'a
  type ('a, 'b) pair = Pair of 'a * 'b
```

Values belonging to such datatypes may be created by applying constructors, and deconstructed with the `match` statement:

```
let rev l acc =
  match l with
    Nil -> acc
  | Cons (h, t) -> rev t (Cons (h, acc))
let reverse l = rev l Nil
```

The first of the mentioned restrictions is to the form of patterns in the `match` statement. Nested patterns are not permitted, and instead each constructor of a datatype must be matched by exactly one pattern. Patterns are also not permitted in the arguments of function definitions. These features must be simulated by nested `match` statements.

The second restriction is on function application. While, function application is written using a curried syntax as above, higher order functions are not permitted in the current version of Camelot. Functions must always be fully applied, and there is no lambda term. This is because closures would seem to introduce an additional non-transparent memory usage, although hopefully this can be overcome at a later date, and higher order functions added to the language.

This basic core is extended with features allowing control of heap allocated storage. The representation of Camelot datatypes is critical here—values from user-defined datatypes are represented by heap-allocated objects from a certain Java class. Consider the earlier list-reversal example. This function allocates an amount of heap-space equal to that occupied by the original list. This space may be eventually reclaimed by the Java garbage collector, but there is no guarantee when this will happen, if at all.

In order to allow better control of heap usage, Camelot includes a *diamond type* (denoted by `<>`) representing regions of heap-allocated memory, and Camelot allows explicit manipulation of diamond objects. These diamond objects correspond directly to objects of the Java diamond class used to representd datatypes. Constructors and match rules are equipped with annotations referring to diamond values:

```
let rev l acc =
  match l with
    Nil -> acc
  | Cons (h, t)@d -> rev t (Cons (h, acc)@d)
let reverse l = rev l Nil
```

The annotation `@d` on the occurrence of `Cons` in the match rule indicates that the space used in that list cell should be made available for re-use via the diamond value `d`. The annotation on the second occurrence of `Cons` specifies that the list cell `Cons(h, acc)` should be constructed in the heap space referred to by `d`.

With explicit management of heap-space comes the possibility of program errors. For example, if the above list reversal function is applied to a sublist of a larger list, the sublist will be correctly reversed but the larger list will become damaged. Various type systems can be used to ensure that diamond annotations are safe. Most simply, we can require all uses of heap-allocated storage to be linearly typed as in [4], but we can also take a less restrictive approach as described in [6]. It is also possible to infer some diamond annotations, as in [5].

1.3 EXTENSIONS

In designing an object system for Camelot, many choices are made for us, or at least tightly constrained. We wish to create a system allowing interoperation with Java, and we wish to compile an object system to JVML. So we are almost forced into drawing the object system of the JVM (and Grail) up to the Camelot level. At least we cannot seriously consider a fundamentally different system.

On the other hand, the type system is strongly influenced by the existing Camelot type system. There is more scope for choice, but implementation can become complex, and an overly complex type system is undesirable from a programmer's point of view. We also do not want to interfere with type systems for resources as mentioned above.

We shall first attempt to make the essential features of Java objects visible in Camelot in a simple form, with the view that a simple abbreviation or module system can be added at a later date to make things more palatable if desired.

Basic Features

We shall view objects as records of possibly mutable fields together with related methods, although Camelot has no existing record system. We define the usual operations on these objects, namely object creation, method invocation, field access and update, and casting and matching. As one might expect we choose a class-based system closely modelling the Java object system. Consequently we must acknowledge Java's uses of classes for encapsulation, and associate static methods and fields with classes also. We now consider these features.

Static method calls There is no conceptual difference between static methods and functions, ignoring the use of classes for encapsulation, so we can treat static method calls just like function calls.

```
java.lang.Math.max a b
```

Static field access Some libraries require the use of static fields. We should only need to provide access to constant static fields, so they correspond to simple values.

```
java.math.BigInteger.ONE
```

Object creation We clearly need a way to create objects, and there is no need to deviate from the `new` operator. By analogy with standard Camelot function application syntax (i.e. curried form) we have:

```
new java.math.BigInteger ‘‘1’’
```

Method invocation Drawing inspiration from the OCaml syntax, and again using a curried form, we have instance method invocation:

```
n#multiply r
```

It could be argued that in order to look like Java, we should not use a curried form, but then there is little reason for the curried form in the core language in any case, lacking higher order functions.

As usual, the arguments of a method invocation may be subclasses of the method's argument types, or classes that implement the specified interface.

Instance field access To retrieve the value of an instance variable, we write

```
object#field
```

whereas to update that value we use the syntax

```
object#field <- value
```

assuming that `field` is declared to be a *mutable* field.

It could be argued that allowing unfettered external access to an object's variables is against the spirit of OO, and more to the point inappropriate for our small language extension, but we wish to allow easy interoperability with any external Java code.

Casts and typecase It shall be necessary to cast objects up to superclasses, and recover subclasses. Here we propose a simple notation for up-casting:

```
obj :> Class
```

This notation is that of OCaml, also borrowed by MLj, and like MLj we shall extend patterns in the manner of `typecase` as follows:

```
match obj with o :> C1 -> o.a()
             | o :> C2 -> o.b()
             | _ -> obj.c()
```

Here o is bound in the appropriate subexpressions to the object obj viewed as an object of type C1 or C2 respectively. The default case is mandatory as in datatype matches.

Unlike MLj we choose not to allow downcasting outside of the new form of match statement, partly because at present Camelot has no exception support to handle invalid down-casts.

An example of the above features follows.

```
let fac′ (n: java.math.BigInteger) r =
  let zero = new java.math.BigInteger "0" in
  let one = java.math.BigInteger.ONE in
  if n#equals zero then r
  else fac′ (n#subtract one) (n#multiply r)
let fac n = fac′ n (java.math.BigInteger.ZERO)
```

The reader may notice a type constraint on the parameter *n*, which we will discuss later. Another example, illustrating the ease of interoperability:

```
let convert (l: string list) =
  match l with [] -> new java.util.LinkedList ()
             | h::t ->
               let ll = convert t
               in let _ = ll#addFirst h
               in ll
```

Defining classes

Once we have the ability to write and compile programs using objects, we may as well start writing classes in Camelot. We must be able to create classes to implement callbacks, such as in the Swing GUI system which requires us to write stateful adaptor classes. Otherwise, as mentioned previously, we may wish to write Camelot code to be called from Java, for example to create a resource-certified library for use in a Java program, and defining a class is a natural way to do this. Implementation of these classes will obviously be tied to the JVM, but the form these take in Camelot has more scope for variation.

We allow the programmer to define a class which may explicitly subclass another class, and implement a number of interfaces. We also allow the programmer to define (possibly mutable) fields and methods, as well as static methods and fields for the purpose of creating a specific class for interfacing with Java. We naturally allow reference to self, and also allow overridden methods from the superclass to be invoked using the super notation as in Java.

Lastly, we allow variables to be written at the top of the class definition, to be bound at object construction and available throughout the body of the object (as in OCaml and MLj). This seems more attractive than an explicit object constructor or *maker*, but we still need to acknowledge that Java objects may have multiple constructors.

The form of a class declaration is:

`class` $\mathit{ClassName}\ x_1 \ldots x_n = \mathit{super}?$
    `implements` $\mathit{Intf}, \ldots, \mathit{Intf}$
    $\mathit{valdec} \ldots \mathit{valdec}$
    $\mathit{method} \ldots \mathit{method}$
`end`

This defines a class called *ClassName*, which must be initialised by calling `new` with n arguments. These values become bound to x_i in the body of the class. The optional *super* clause takes the form:

$\mathit{SuperClassName}\ y_1 \ldots y_m$ `with`

This causes *ClassName* to be a direct subclass of *SuperClassName*. Additionally, the superclass is constructed with arguments y_i, as if by the expression `new` $\mathit{SuperClassName}\ y_1 \ldots y_m$. The arguments $y_1 \ldots y_m$ must be a subset of the bound variables $x_1 \ldots x_n$. *ClassName* inherits the methods and values present in its superclass, and these may be referred to in its definition. If no superclass is given, the class will be a subclass of `java.lang.Object`, the root of the class hierarchy.

As well as a superclass, a class can declare that it implements one or more interfaces. These correspond directly to the Java notion of an interface. Java libraries often require the creation of a class implementing a particular interface—for example, to use a Swing GUI one must create classes implementing various interfaces to be used as callbacks. Note that at the current time it is not possible to define interfaces in Camelot, they are provided purely for the purpose of interoperability.

Now we describe field declarations:

`field` $x = e$ `val` $x = e$
`field mutable` $x = e$

Instance fields are defined using the keyword `field`, and can be declared to be mutable or not, but in either case they must be initialised at the point of definition. As mentioned earlier, the variables bound at the top of the class definition are available for use in the expression on the right hand side of the definition. It is also possible to define static fields using `val`.

Methods are defined as follows:

`method` $m\ x_1 \ldots x_n$ = e `fun` $m\ x_1 \ldots x_n$ = e
`method` m `()` = e `fun` m `()` = e

Again, notice that we use the usual `fun` syntax to declare what Java would call static methods. Static methods are simply *monomorphic* Camelot functions which happen to be defined within a class, and are used as described previously. Instance methods, on the other hand, are actually a fundamentally new addition to the language. We consider the instance methods of a class to be a set of mutually recursive monomorphic functions, in which the special variable `this` is bound to the current object of that class.

We can consider the methods as mutually recursive without using any additional syntax (such as `and` blocks) since they are monomorphic, and as such we do not risk preventing generalisation of type variables. In any case this implicit mutual recursion feels appropriate when we are compiling to the Java Virtual Machine, and have to come to terms with open recursion in any case.

In the body of an instance method, the syntax

`super#`$m\ x_1 \ldots x_n$

is used to invoke the method m defined in the superclass, ignoring any overriding definition in the current class. There is a good case that this `super` syntax is confusing, as it does not simply refer to the object as if it were cast to the superclass, but while there may be subtlety in the semantics of `super` we feel that this still reflects the user's intuition.

1.4 IMPLEMENTATION

Typing

The typing rules in Figure 1.1 are for the Object Oriented extensions to Camelot. Typing rules for the base language are roughly as one might expect, except that the rule for function application forces functions to be fully applied.

Consider rules INVOKE, and FIELD. Firstly, types must match exactly for field access, whereas methods can be called with subtypes of their argument types. Otherwise these are fairly similar.

Secondly, note that we look up methods(c) (respectively fields(c)). This actually implies that at the time this rule is applied the class of the object in question must be known, at least in the obvious translation to a constraint-typing rule. This has real consequences for the programmer—the programmer must ensure that the type of the object is suitably constrained at the time of invocation. In practise, this will probably mean that almost all function arguments of object type must be constrained before use, and coercions may also be necessary in some places. In such a system, it is probably not feasible to resolve method overloading cleanly.

$$\textsc{Invoke}\ \frac{\Sigma \vdash e : c \quad (id : \tau_1 \to \ldots \to \tau_n \to \tau) \in \text{methods}(c) \quad \Sigma \vdash x_i : \tau_i' \quad \tau_i' \leqslant \tau_i}{\Sigma \vdash e\#id\ x : \tau}$$

$$\textsc{Field}\ \frac{\Sigma \vdash e : c \quad (id : \tau) \in \text{fields}(c)}{\Sigma \vdash e\#id : \tau}$$

$$\textsc{Cast}\ \frac{\Sigma \vdash e : \tau \quad \tau \leqslant \tau'}{\Sigma \vdash e :> \tau' : \tau'}$$

FIGURE 1.1. Additional Camelot typing rules

A more intelligent solution would only place constraints to be solved globally, but unfortunately these cannot be equality constraints, and so we have to depart from the simple unification algorithm. We are not alone in this problem. The MLj implementation also suffers from this. In [1], it is noted that constraints must be solved globally for a more usable implementation, but the existing implementation did not do this. In [9], a new type inference algorithm is given for MLj which uses branching search and backtracking. Branching search is required because of the complexities of the type system, including implicit coercions such as `option`, and it may be that our more naïve type system could use a simpler algorithm.

A solution avoiding these issues may be to avoid considering method invocations during type inference. Constraints could be inferred and solved by unification as usual, but with no constraints present for these invocations. After unification has taken place, we will be left with a typed program with some free type variables, and we can then resolve overloading in a more simplistic fashion (as the types of objects and method arguments should be known by this point). The remaining type variables will thus be instantiated after unification.

Translation

As mentioned earlier, the target for the present Camelot compiler is Java bytecode. However we make use of the intermediate language Grail (see [2]). Grail is a low-level functional language, and is basically a functional form for Java bytecode. Grail's functional nature makes the compilation from Camelot more straightforward, but Grail is faithful enough to JVML that the compilation process is reversible.

Here we use the notation of Grail to describe the compilation of new Camelot features, but mostly the meanings of Grail phrases should be self-evident. However, it is worthwhile noting that the JVML basic blocks comprising a Camelot method are represented in Grail by a collection of mutually tail-recursive functions—

Camelot expression	Grail operation
`java.lang.Math.max a b`	`invokestatic <int java.lang.Math.max (int,int)> (a, b)`
`java.math.BigInteger.ONE`	`getstatic <java.math.BigInteger java.math.BigInteger.ONE>`
`new java.math.BigInteger ''1''`	`new <java.math.BigInteger (java.lang.String)> ("1")`
`n#multiply r`	`invokevirtual <java.math.BigInteger java.math.BigInteger.multiply (java.math.BigInteger)> (r)`
`object#field`	`getfield object <type TheClass.field>`
`obj :> Class`	`checkcast Class obj`

TABLE 1.1. Translation of new Camelot expressions

calling these functions corresponds to JVML jump instructions. Also there are several different method invocation instructions: `invokestatic` for static methods (as we translate standard Camelot functions to), `invokevirtual` for instance methods, and `invokespecial` for calling object constructors. Grail differs from JVML by combining object creation and initialisation into the `new` instruction, but we must still in one case use the `invokespecial` instruction.

Notational issues aside, translating the new features is relatively straightforward, as the JVM (and Grail) provide what we need. Table 1.1 gives the translations of many features.

The new match expressions are more complex. An example of the new type of match statement is

`match` e `with`
    o_1 `:>` C_1 `->` e_1
    $\vdots$
    o_n `:>` C_n `->` e_n

where each C_i is a class name. We generate functions as in Figure 1.2. Additionally we generate functions $\rho_1 \ldots \rho_n$ which compute the expressions $e_1 \ldots e_n$ then proceed with the current computation.

Making Classes

Compiling new classes is fairly straightforward. Value definitions translate to instance variables, and method definitions are method definitions. Since a method is not so far from a function, the machinery required to translate methods is almost all there; references to self must also be allowed, and the correct scope used.

```
fun $\beta_1(\ldots)$ =
let
   val $i_1$ = instance $C_1$ $e$
in
   if $i_1$ = 1 then $\gamma_1$ else $\beta_2$
end

fun $\gamma_1$ $(\ldots)$ =
let
   val $o_1$ = checkcast $C_1$ $e$
in $\rho_1$ end
```

...

```
fun $\beta_{n-1}(\ldots)$ =
let
   val $i_{n-1}$ = instance $C_{n-1}$ $e$
in
   if $i_{n-1}$ = 1 then $\gamma_{n-1}$ else $\gamma_n$
end

fun $\gamma_n$ $(\ldots)$ =
let
   val $o_n$ = checkcast $C_n$ $e$
in $\rho_n$ end
```

...

FIGURE 1.2. Functions generated for `match` expression

Mutable fields must also be considered—these correspond to a `putfield` instruction.

Another aspect is initialisers. The following example sketches a translation.

```
class X x y z = Y x y with
    val a = z + 1
    val b = y
    method ...
end

class X {
  field int a

  method <init> (int x, int y, int z) =
  let
    val () = invokespecial <Y.<init>(int, int)> (x, y)
    val t = z+1
  in
    putfield this <int X.a> t
  end

  method ...
}
```

1.5 RELATED WORK

We have made reference to MLj, the aspects of which related to Java interoperability are described in [1]. MLj is a fully formed implementation of Standard ML, and as such is a much larger language than we consider here. In particular, MLj can draw upon features from SML such as modules and functors, for example, allowing the creation of classes parameterised on types. Such flexibility comes with a price, and we hope that the restrictions of our system will make the certification of the resource usage of O'Camelot programs more feasible.

By virtue of compiling an ML-like language to the JVM, we have made many of the same choices that have been made with MLj. In many cases there is one obvious translation from high level concept to implementation, and in others the appropriate language construct is suggested by the Java object system. However we have also made different choices more appropriate to our purpose, in terms of transparency of resource usage and wanting a smaller language. For example, we represent objects as records of mutable fields whereas MLj uses immutable fields holding references.

There have been various other attempts to add object oriented features to ML and ML-like languages. OCaml provides a flexible system with many features (a formalised subset is described in [10]), but at the cost of a complex type system. While it may be possible to compile such a system to JVML, it would be very hard to define a simple interlanguage interface. A class mechanism for Moby is defined in [3] with the principal that classes and modules should be orthogonal concepts. Lacking a module system, Camelot is unable to take such an approach, but both Moby and OCaml have been a guide to concrete representation.

Many other relevant issues are discussed in [8], but again Camelot's lack of a module system (and our desire to avoid this to keep the language small) gives us a different perspective on the issues.

1.6 CONCLUSION

We have described the language Camelot and its unique features enabling the control of heap-allocated data, and outlined an object-oriented extension allowing interoperability with Java programs and libraries. We have kept the language extension fairly minimal in order to facilitate further research on resource aware programming, yet it is fully-featured enough for the mobile applications we envisage for Camelot.

The O'Camelot compiler is a work in progress, but implements almost all the features described here. The current version of the compiler can be obtained from `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

REFERENCES

[1] Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with java. In *International Conference on Functional Programming*, pages 126–137, 1999.

[2] L. Beringer, K. MacKenzie, and I.G. Stark. Grail: a functional form for mobile imperative code.

[3] K. Fisher and J. Reppy. Moby objects and classes, 1998.

[4] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[5] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, January 2003.

[6] Michal Konečný. Typing with conditions and guarantees in LFPL. In *Types for Proofs and Programs: Proceedings of the International Workshop TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2002.

[7] K. MacKenzie and N. Wolverson. Camelot and Grail: Compiling a resource-aware functional language for the java virtual machine. Submitted to *Implementation of Functional Languages 2003*.

[8] David MacQueen. *Formal Aspects of Computing*, 2002.

[9] Bruce McAdam. Type inference for mlj. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 2. Intellect, 2000.

[10] Didier Remy and Jerome Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.