Chapter 4

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

Nicholas Wolverson and Kenneth MacKenzie[1]

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, harnessing the power of Java libraries and allowing Java programs to incorporate resource-bounded Camelot code.[2]

4.1 INTRODUCTION

The Mobile Resource Guarantees (MRG) project aims to equip mobile bytecode programs with guarantees that their usage of certain computational resources (such as time, heap space 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 [8], enables the programmer to

[1]Laboratory for Foundations of Computer Science, The University of Edinburgh. Email: N.Wolverson@ed.ac.uk, kwxm@inf.ed.ac.uk

[2]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.

write a program with a predictable resource usage; future work will provide each program with a proof that it does not exceed a stated resource bound. A compiler exists for this language, compiling polymorphic resource-aware Camelot programs to the JVM. However, only primitive interaction with the outside world is possible, through command line arguments, file input 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 to interface with external Java libraries. Similarly, the programmer may wish to utilise device-specific libraries or 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.

4.2 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.

The concrete syntax of Camelot is very close to O'Caml, as described in [1]. The following program defines a polymorphic list datatype and functions `sort` and `insert` performing an insertion sort on such lists.

```
type 'a lst = !Nil | Cons of 'a * 'a lst
let rec insert n l d =
  match l with Nil -> Cons(n, Nil)@d
             | Cons(h,t)@d' ->
               if n <= h then Cons(n, Cons(h,t)@d')@d
               else Cons(h, insert n t d)@d'
let rec sort l =
  match l with Nil -> Nil
             | Cons(h,t)@d -> insert h (sort t) d
```

Ignoring annotations such as `@d` and occurrences of the associated variable `d`, and the `!` in front of `Nil`, this program is valid O'Caml and indeed defines an insertion sort. Here we are more concerned about space rather than time issues; notice that the datatype constructor `Cons` is applied $O(n^2)$ times on average, but this much storage is not necessary. While a sensible garbage collector means we will not really lose the use of this space, this is not guaranteed, and we cannot predict when the space will be reclaimed. This is unacceptable when considering proof carrying code, and indeed on some mobile devices we will not have the luxury of a garbage collector at all.

In order to allow better control of heap usage, Camelot adds features allowing control of heap allocated storage. Camelot includes a *diamond type* (denoted by

<>) representing regions of heap-allocated memory and allows explicit manipulation of diamond objects. The representation of Camelot datatypes is critical here—values from user-defined datatypes are represented by heap-allocated objects from a certain Java class and a diamond value corresponds directly to an object of this class.

The diamond annotations in the above program result in an in-place insertion sort algorithm. During the execution of `sort` on a list, no new block of heap storage is allocated, but instead the existing storage is reused for the new list. The annotation `@d` on the occurrence of `Cons` in `sort` indicates that the space used in that list cell should be made available for re-use via the diamond value `d`. This diamond value is passed to a call of `insert`, where it is used in the expression `Cons(n, Nil)@d` to specify that the cons cell should be constructed in the heap space referred to by `d`. Lastly the use of `!` in the definition of the `Nil` constructor indicates that `Nil` does not take up a diamond (`Nil` is represented by the null pointer).

With explicit management of heap-space comes the possibility of program errors. The above sort function destroys the original list, so any subsequent attempt to reuse that list may result in an error, and if the list is a sublist of a larger list, the sublist will be correctly sorted 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 described in [5]; the above program is typable under this system. We can also take a less restrictive approach as described in [7]. It is also possible to infer some diamond annotations, as shown in [6], and indeed this process can also give an upper bound on a program's heap usage.

As well as adding resource-related extensions, we make some restrictions, the first of which 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.

4.3 EXTENSIONS

In designing an object system for Camelot, many choices are made for us, or are at least tightly constrained. We wish to create a system allowing inter-operation 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 up to the Camelot level and 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. The examples below illustrate the new classes of expressions we add to Camelot.

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 "101010" 2
```

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.

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

```
myMap#put key value
```

Null values In Java, any method with object return type may return the `null` object. For this reason we add a construct

`isnull` *e*

which tests if the expression *e* is a `null` value.

Casts and typecase It may occasionally be necessary to cast objects up to superclasses, for example to force the intended choice between overloaded methods. We will also want to recover subclasses, such as when removing an object from a collection. Here we propose a simple notation for up-casting:

```
obj :> Class
```

This notation is that of O'Caml, also borrowed by MLj (described in [2]). To handle down-casting we shall extend patterns in the manner of `typecase` (again like MLj) 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. As in datatype matches, we require that every possible case is covered; here this means that the default case is mandatory. We also require that each class is a subclass of the type of `obj`, and suggest that a compiler warning should be given for any redundant 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.

As usual, the arguments of a (static or instance) method invocation may be subclasses of the method's argument types, or classes implementing the specified interfaces.

The following example demonstrates some of the above features and illustrates the ease of interoperability. We will discuss the need for type constraints as on the parameter *l* later.

```
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 `this`.

The form of a class declaration is given below. Items within angular brackets $\langle \ldots \rangle$ are optional.

$$\begin{array}{rcl} \mathit{classdecl} & ::= & \texttt{class}\ \mathit{cname}\ \texttt{=}\ \langle \mathit{scname}\ \texttt{with} \rangle\ \mathit{body}\ \texttt{end} \\ \mathit{body} & ::= & \langle \mathit{interfaces} \rangle\ \langle \mathit{fields} \rangle\ \langle \mathit{methods} \rangle \\ \mathit{interfaces} & ::= & \texttt{implement}\ \mathit{iname}\ \langle \mathit{interfaces} \rangle \\ \mathit{fields} & ::= & \mathit{field}\ \langle \mathit{fields} \rangle \\ \mathit{methods} & ::= & \mathit{method}\ \langle \mathit{methods} \rangle \end{array}$$

This defines a class called *cname*, implementing the specified interfaces. The optional *scname* gives the name of the direct superclass; if it is not present, the superclass is taken to be the root of the class hierarchy, namely `java.lang.Object`. The class *cname* inherits the methods and values present in its superclass, and these may be referred to in its definition.

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.

$$\mathit{field} \quad ::= \quad \texttt{field}\ x : \tau \mid \texttt{field}\ \texttt{mutable}\ x : \tau \mid \texttt{val}\ x : \tau$$

Instance fields are defined using the keyword `field`, and can optionally be declared to be mutable. Static fields are defined using `val`, and are non-mutable. In a sense these mutable fields are the first introduction of side-effects into Camelot. While the Camelot language is defined to have an array type, this has largely been ignored in our more formal treatments as it is not fundamental to the language. Mutable fields, on the other hand, are fundamental to our notion of object

orientation, so we expect any extension of Camelot resource-control features to O'Camelot to have to deal with this properly.

Methods are defined as follows, where $1 \leq i_1 \ldots i_m \leq n$.

$$
\begin{array}{rcl}
\mathit{method} & ::= & \texttt{maker}(x_1 \texttt{:} \tau_1) \ldots (x_n \texttt{:} \tau_n) \langle \texttt{: super}\ x_{i_1} \ldots x_{i_m} \rangle = \mathit{exp} \\
& & \mid \texttt{method}\ m(x_1 \texttt{:} \tau_1) \ldots (x_n \texttt{:} \tau_n) \texttt{ : } \tau = \mathit{exp} \\
& & \mid \texttt{method}\ m\texttt{() : } \tau = \mathit{exp} \\
& & \mid \texttt{let}\ m(x_1 \texttt{:} \tau_1) \ldots (x_n \texttt{:} \tau_n) \texttt{ : } \tau = \mathit{exp} \\
& & \mid \texttt{let}\ m\texttt{() : } \tau = \mathit{exp}
\end{array}
$$

Again, we use the usual `let` syntax to declare what Java would call static methods. Static methods are simply *monomorphic* Camelot functions which happen to be defined within a class, although they are invoked using the syntax described earlier. 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. ML uses `and` blocks to group mutually recursive functions because its *let-polymorphism* prevents any of these functions being used polymorphically in the body of the others, but this is not an issue here. 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 addition to static and instance methods, we also allow a special kind of method called a *maker*. This is just what would be called a constructor in the Java world, but as in [4] we use the term maker in order to avoid confusion between object and datatype constructors. The `maker` term above defines a maker of the containing class C such that if `new` C is invoked with arguments of type $\tau_1 \ldots \tau_n$, an object of class C is created, the superclass maker is executed (this is the zero-argument maker of the superclass if none is explicitly specified), expression *exp* (of `unit` type) is executed, and the object is returned as the result of the `new` expression. Every class has at least one maker; a class with no explicit maker is taken to have the maker with no arguments which invokes the superclass zero-argument maker and does nothing. This implicit maker is inserted by the compiler.

4.4 TYPING

Typing rules for some of the more important Object Oriented extensions are given in Fig. 4.1. Rules for static method invocation and static field access are similar to those given for instance versions, and rules for the base language are roughly as one might expect, except that the rule for function application forces functions to be fully applied. The requirement above to state the types of fields, methods and makers at the point of definition means we can easily construct the sets of these types as makers(C), methods(C) and fields(C) for each class C.

$$\textsc{New}\ \frac{(\tau_1 \to \ldots \to \tau_n) \in \text{makers}(C) \quad \Sigma \vdash x_i : \tau_i' \quad \tau_i' \leqslant \tau_i}{\Sigma \vdash \texttt{new}\, C\, x_1 \ldots x_n : C}$$

$$\textsc{Invoke}\ \frac{\Sigma \vdash e : C \quad (id : \tau_1 \to \ldots \to \tau_n \to \tau) \in \text{methods}(C) \quad \Sigma \vdash x_i : \tau_i' \quad \tau_i' \leqslant \tau_i}{\Sigma \vdash e\#id\ x_1 \ldots x_n : \tau}$$

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

$$\textsc{Update}\ \frac{\Sigma \vdash e : C \quad (id : \tau) \in \text{fields}(C) \quad \Sigma \vdash e' : \tau}{\Sigma \vdash e\#id\ \texttt{<-}\ e' : \texttt{unit}}$$

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

FIGURE 4.1 Additional Camelot typing rules

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 implies that at the time this rule is applied the class C of the object in question must be known, at least in the obvious implementation. 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 practice, 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.

Additionally, method (and maker) overloading introduces ambiguity. Different instances of INVOKE or NEW may apply depending on the argument types, and indeed for many argument types there is no unique applicable method. In Java this is resolved by choosing the "most specific" method if it exists. In combination with the standard type inference algorithm this forces us to know the type of all arguments to a method at the point it is applied. Indeed in our current implementation this is exactly what happens; we assume argument types are available at the point of application and compute the most specific of the applicable methods. Again this puts a burden on the programmer, although in practice this has been proved in reasonable examples.

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; for example, the MLj implementation described in [2] also suffers from this. In [10], a new type inference algorithm is given for MLj which solves a system of more complex constraints using 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 naive type system could use a simpler algorithm.

One way of avoiding these issues could 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. Unfortunately this resolution requires another full typechecking, and indeed in our present implementation it may be easier to implement a system in the style of [10] if necessary.

Polymorphism

We remarked earlier that static methods are basically monomorphic Camelot functions together with a form of encapsulation, but it is worth considering polymorphism more explicitly. O'Camelot methods, whether static or instance methods, are not polymorphic. That is, they have subtype polymorphism but not parametric polymorphism (genericity), unlike Camelot functions which have parametric but not subtype polymorphism. This is not generally a problem, as most polymorphic functions will involve manipulation of polymorphic datatypes and can be placed in the main program, whereas most methods will be interfacing with the Java world and thus should conform to Java's subtyping polymorphism.

4.5 TRANSLATION

As mentioned earlier, the target for the present Camelot compiler is Java bytecode. However we make use of the intermediate language Grail (see [3]). 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—calling these functions corresponds to JVML goto instructions. There are several different method invocation instructions, namely `invokestatic` for static methods, `invokevirtual` for instance methods, and `invokespecial` for calling object constructors—standard Camelot functions are tranlsated to static methods, and their application correpsonds to an `invokestatic` instruction. Grail differs from JVML by combining object creation and initialisation into the `new` instruction, but we must still use the `invokespecial` instruction to call the superclass constructor.

Notational issues aside, translating the new features is relatively straightforward, as the JVM (and Grail) provide what we need. In particular, Grail is suf-

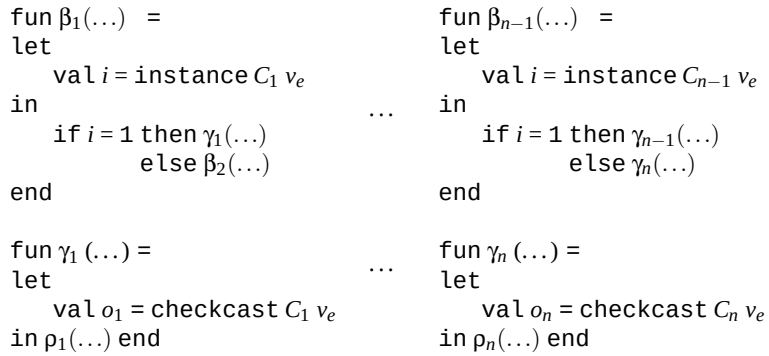

FIGURE 4.2 Functions generated for `match` expression

ficiently expressive that it was not necessary to extend the compiler backend significantly.

Function ϕ below informally specifies the translation of the new Camelot expressions to Grail code. We assume these expressions are normalised in the style of the basic Camelot expressions, so that all subexpressions are atomic and have a simple Grail expansion, rather than requiring the generation of extra Grail functions and let statements.

ϕ(*package.Class.method* $x_1 \ldots x_n$) =
  `invokestatic <`τ_{ret} *package.Class.method* τ_{arg}`> (`$\phi(x_1), \ldots, \phi(x_n)$`)`
ϕ(*package.Class.field*) = `getstatic <`τ *package.Class.field*`>`
ϕ(`new` *package.Class* $x_1 \ldots x_n$) = `new <`*package.Class*(τ_{arg})`> (`$\phi(x_1) \ldots \phi(x_n)$`)`
ϕ(*obj#mname* $x_1 \ldots x_n$) =
  `invokevirtual` *obj* `<`τ_{ret} *package.Class.mname* (τ_{arg})`> (`$\phi(x_1) \ldots \phi(x_n)$`)`
ϕ(*obj#field*) = `getfield` *obj* `<`τ *package.Class.field*`>`
ϕ(*obj#field*`<-`*exp*) = `putfield` *obj* `<`τ *package.Class.field*`>` *exp*
ϕ(*obj* `:>` *package.Class*) = `checkcast` *package.Class obj*
ϕ(`isnull` *exp*) = *exp* `= null[`τ`]`

Types τ, τ_{arg} and τ_{ret} are Grail types derived from the Camelot types inferred for the appropriate fields and methods. To illustrate the above translation, we show the translation of the multiplication of two `BigInteger` objects using the `multiply` instance method.

```
φ(n#multiply r) =
   invokevirtual n <java.math.BigInteger
   java.math.BigInteger.multiply
   (java.math.BigInteger)> (r)
```

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

of match statement is

```
match e with
      o1 :> C1 -> e1
      ⋮
      on :> Cn -> en
```

where each C_i is a class name. We generate functions as in Fig 4.2, where β_1 will be the first function to be executed, i is a fresh variable, and v_e is a variable holding the result of evaluating expression e. 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

Translating class definitions is fairly straightforward. A `val` declaration corresponds to a final static field, the type of which is the translation of the stated Camelot type. Similarly a `field` definition corresponds to an instance field of the appropriate type, which will be `final` if the field is not `mutable`.

A `maker` corresponds to a method called `<init>` taking arguments of the appropriate type (returning `void`), and calling the appropriate `<init>` method in the superclass before executing the code corresponding to expression in the body, which is compiled as above.

As remarked earlier, static methods are basically monomorphic Camelot functions encapsulated in a class, and so their compilation is just as standard Camelot functions. Instance methods are also compiled like monomorphic Camelot functions, but references to `this` are permitted.

4.6 OBJECTS AND RESOURCE TYPES

As described in Sec. 4.2, the use of diamond annotations on Camelot programs in combination with certain resource-aware type systems allows the heap usage of those programs to be inferred, as well as allowing some in-place update to occur. Clearly the presence of mutable objects in O'Camelot also provides for in-place update. However by allowing arbitrary object creation we also replicate the unbounded heap-usage problem solved for datatypes. Perhaps more seriously, we are allowing Camelot programs to invoke arbitrary Java code, which may use an unlimited amount of heap space.

First consider the second problem. Even if we have some way to place a bound on the heap space used by our new OO features within a Camelot program, external Java code may use any amount of heap whatsoever. There seem to be a few possible approaches to this problem, none of which are particularly satisfactory. We could decide only to allow the use of external classes if they came with a proof of bounded heap usage. Constructing a resource-bounded Java class library or inferring resource bounds for an existing library would be a massive undertak-

ing, although perhaps less problematic with the smaller class libraries used with mobile devices. This suggestion seems somewhat unrealistic.

Alternatively, we could simply allow the resource usage of external methods to be stated by the programmer or library creator. This extends the trusted computing base in the sense of resources, but seems a more reasonable solution. The other alternative—considering resource-bound proofs only to refer to the resources directly consumed by the Camelot code—seems unrealistic, as one could easily (and even accidentally) cheat by using Java libraries to do some memory-consuming "dirty work".

The issue of heap-usage *internal* to O'Camelot programs seems more tractable, although we do not propose a solution here. A first attempt might mimic the techniques used earlier for datatypes; perhaps we can adapt the use of diamonds and linear type systems? The use of diamonds for in-place update is irrelevant here and indeed relies on the uniform representation of datatypes by objects of a particular Java class. Since we are hardly going to represent every Java object by an object of one class we could not hope to have such a direct correlation between diamonds and chunks of storage.

However, we could imagine an abstract diamond which represents the heap storage used by an arbitrary object and require any instance of `new` to supply one of these diamonds, in order that the total number of objects created is limited. Unfortunately reclamation of such an abstract diamond would only correspond to making an object available to garbage collection, rather than definitely being able to re-use the storage. Even so, such a system might be able to give a measure of the total number of objects created and the maximum number in active use simultaneously.

4.7 RELATED WORK

We have made reference to MLj, the aspects of which related to Java interoperability are described in [2]. 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. O'Caml provides a clean, flexible object system with

many features and impressive type inference—a formalised subset is described in [12]. As in O'Camelot, objects are modelled as records of mutable fields plus a collection of methods. Many of the additional features of O'Caml could be added to O'Camelot if desired, but there are some complications caused when we consider Java compatibility. For example, there are various ways to compile parameterised classes and polymorphic methods for the JVM, but making these features interact cleanly with the Java world is more subtle.

The power of the O'Caml object system seems to come more from the distinctive type system employed. O'Caml uses the notion of a *row variable*, a type variable standing for the types of a number of methods. This makes it possible to express "a class with these methods, and possibly more" as a type. Where we would have a method parameter taking a particular object type and by subsumption any subtype, in O'Caml the type of that parameter would include a row variable, so that any object with the appropriate methods and fields could be used. This allows O'Caml to preserve type inference, but this is less important for our application and does not map cleanly to the JVM.

A class mechanism for Moby is defined in [4] with the principle that classes and modules should be orthogonal concepts. Lacking a module system, Camelot is unable to take such an approach, but both Moby and O'Caml have been a guide to concrete representation. Many other relevant issues are discussed in [9], but again lack of a module system—and our desire to avoid this to keep the language small—gives us a different perspective on the issues.

4.8 CONCLUSION

We have described the language Camelot and its unique features enabling the control of heap-allocated data and have 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 implements all the features described here. The current version of the compiler can be obtained from

`http://www.lfcs.inf.ed.ac.uk/mrg/camelot/`

A EXAMPLE

Here we give an example of the features defined above. The code below, together with the two standard utility functions `rev` and `len` for list reversal and length, defines a program for Sun's MIDP platform (as described in [11]), which runs on devices such as PalmOS PDAs. The program displays the list of primes in an interval. Two numbers are entered into the first page of the GUI, and when a button is pressed a second screen appears with the list of primes, calculated using the sieve of Eratosthenes, along with a button leading back to the initial display.

This example has been compiled with our current compiler implementation, and executed on a PalmOS device.

```
class primes = javax.microedition.midlet.MIDlet with
  implement javax.microedition.lcdui.CommandListener

  field exitCommand: javax.microedition.lcdui.Command
  field goCommand: javax.microedition.lcdui.Command
  field doneCommand: javax.microedition.lcdui.Command
  field mainForm: javax.microedition.lcdui.Form
  (* lower and upper limits: *)
  field lltf: javax.microedition.lcdui.TextField
  field ultf: javax.microedition.lcdui.TextField
  field display: javax.microedition.lcdui.Display

  maker () =
    let _ = display <-
      (javax.microedition.lcdui.Display.getDisplay
        (this:> javax.microedition.midlet.MIDlet))
    in let _ = goCommand <-
      (new javax.microedition.lcdui.Command
      "Go" javax.microedition.lcdui.Command.SCREEN 1)
    in let _ = exitCommand <-
      (new javax.microedition.lcdui.Command
      "Exit" javax.microedition.lcdui.Command.SCREEN  2)
    in let t = new javax.microedition.lcdui.Form "Primes"
    in let ll = new javax.microedition.lcdui.TextField
        "Lower limit:" "" 10
        javax.microedition.lcdui.TextField.NUMERIC
    in let _ = lltf <- ll
    in let _ = t#append ll
    in let ul = new javax.microedition.lcdui.TextField
        "Upper limit:" "" 10
        javax.microedition.lcdui.TextField.NUMERIC
    in let _ = ultf <- ul
    in let _ = t#append ul
    in let _ = t#addCommand (this#goCommand)
    in let _ = t#addCommand (this#exitCommand)
    in let _ = mainForm <- t
    in t#setCommandListener this

  method startApp (): unit =
    this#display#setCurrent (this#mainForm)
  method pauseApp (): unit = ()
  method destroyApp (b:bool): unit = ()
  method commandAction
           (cmd: javax.microedition.lcdui.Command)
           (s: javax.microedition.lcdui.Displayable)
           : unit =
```

```
    if cmd#equals (this#exitCommand)
    then let _ = this#destroyApp false
         in this#notifyDestroyed ()
    (* create & display list of primes *)
    else if cmd#equals (this#goCommand)
    then
      let    lower_limit = int_of_string
                           (this#lltf#getString())
      in let upper_limit = int_of_string
                           (this#ultf#getString())
      in let primes =
               new javax.microedition.lcdui.Form "Primes"
      in let _ = appendPrimes lower_limit upper_limit primes
      in let done = new javax.microedition.lcdui.Command
                  "Done"
                  javax.microedition.lcdui.Command.SCREEN 1
      in let _ = doneCommand <- done
      in let _ = primes#addCommand done
      in let _ = primes#setCommandListener this
      in let _ =
        javax.microedition.lcdui.AlertType.INFO#playSound
          (this#display)
      in this#display#setCurrent primes
   (* back to main form *)
    else if cmd#equals (this#doneCommand) then
      this#display#setCurrent (this#mainForm)
    else ()
end
(* Generate a list of prime numbers in an interval [a..b] *)
(* Integer square roots *)
let increase k n = if (k+1)*(k+1) > n then k else k+1
let rec intsqrt n = if n = 0 then 0
                    else increase (2*(intsqrt (n/4))) n

(* n is divisible by no member of l which is <= sqrt n *)
let isPrime n l lim =
  match l with
    [] -> true
  | h::t -> h <= lim && n mod h <> 0 && isPrime n t lim

(* generate list of primes between n and top *)
let make1 n top acc =
  if n > top then rev acc []
  else if isPrime n acc n then make1 (n+2) top (n::acc)
  else make1 (n+2) top acc

let makeSmallPrimes top = make1 3 top [2]
let makePrimes n top smallPrimes =
  if n > top then []
```

```
  else if isPrime n smallPrimes n then
    n::(makePrimes (n+2) top smallPrimes)
  else makePrimes (n+2) top smallPrimes

let appList l (f: javax.microedition.lcdui.Form) =
  match l with [] -> ()
  | (h::t)@_ -> let _ = f#append ( (string_of_int h)^"\n")
                in appList t f

let appendPrimes bot top
      (f: javax.microedition.lcdui.Form) =
  let smallPrimes = makeSmallPrimes (intsqrt top)
  in let primes = makePrimes (bot + 1 - bot mod 2)
                     top smallPrimes
  in let s = (string_of_int (len primes)) ^ " primes\n"
  in let _ = f#append s
  in appList primes f
```

REFERENCES

[1] O'Caml. See `http://www.ocaml.org`.

[2] Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. In *Proc. of ICFP*, pages 126–137, 1999.

[3] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In Vladimiro Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[4] K. Fisher and J. Reppy. Moby objects and classes, 1998. Unpublished manuscript.

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

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

[7] 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.

[8] K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware functional programming for the JVM. In *Trends in Functional Programming Volume 4: Proceedings of TFP2003*, pages 29–46. Intellect, 2004.

[9] David MacQueen. Should ML be object-oriented? *Formal Aspects of Computing*, 13(3-5), 2002.

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

[11] Sun Microsystems. Mobile Information Device Profile (MIDP). See `http://java.sun.com/products/midp/`.

[12] 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.