Chapter 1

Camelot and Grail: Resource-Aware Functional Programming on the JVM[1]

K. MacKenzie and N.Wolverson[2]

Abstract: We describe the functional language Camelot, which is a language of the ML family with extensions for explicit management of heap storage, and the intermediate language Grail, which is a functional form of JVM bytecode. A scheme for transforming Camelot into Grail is described. We also give some figures for execution times which show that Camelot programs perform reasonably well when compared with Java equivalents.

1.1 INTRODUCTION

The Mobile Resource Guarantees (MRG) project [15] aims to develop a Proof Carrying Code (PCC) [16] framework to endow mobile computer programs with guarantees of resource bounds. Typical resources are time, heap space, system calls, and stack size. Our goal is to provide a resource-safe programming language to be used for writing mobile code. This language, which is called Camelot, is a high-level functional language which is compiled into JVM bytecode. The class files produced by the compiler will be equipped with a proof that the programs obey specified resource constraints, and can then be transmitted across a network in the usual way. The consumer of the mobile code can then independently verify the resource constraints by checking the proof attached to the code; if verification

[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.

[2]Laboratory for the Foundations of Computer Science, School of Informatics, The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK; Email: kwxm@inf.ed.ac.uk, N.Wolverson@sms.ed.ac.uk

is successful then execution can proceed as normal. This technique provides an unforgeable guarantee that the claimed resource limits will not be exceeded.

In this paper we will describe Camelot and its translation to JVM bytecode. Camelot is similar to a subset of O'Caml, the main novelty lying in extensions for performing in-place modifications to heap-allocated data-structures. These features are similar to those described in by Hofmann in [6], but include some extra extensions for freelist management. To retain a purely functional semantics for the language in the presence of these extensions a linear type system can be employed: in the present implementation, linearity can be enforced via a compiler switch. We are in the process of enhancing the compiler by the addition of other, less restrictive type systems which still allow safe in-place modifications: more details will be given below.

Crucial design choices for the compilation are transparency and an exact specification of the compilation process. The former ensures that the compilation does not modify the resource consumption in an unpredictable way. The latter provides a formal basis for using resource information inferred for the high-level language in proofs on the intermediate language.

1.2 CAMELOT

Camelot is a strongly typed language of the ML family with features added to enable close control of heap usage. The syntax of Camelot (which is similar to a subset of the syntax of the O'Caml language [17]) is given below. The terms `tycon`, `cname`, `fname` and and `var` refer to *type constructors*, *constructor names*, *function names* and *variable names* respectively: all of these are names in SML style. Constructor names must begin with an upper-case letter, whereas all other identifiers begin with a lower-case letter. The term *tyvar* refers to a *type variable*, which is a name beginning with a single quote. Literal constants (*const* below) are similar to those in O'Caml. Optional items are enclosed in angular parentheses.

$$
\begin{array}{rcl}
\textit{program} & ::= & \langle \textit{typedecseq} \rangle\ \langle \textit{valdecseq} \rangle\ \langle \textit{funimpseq} \rangle \\
\textit{typedecseq} & ::= & \textit{typedec}\ \langle \textit{typedecseq} \rangle \\
\textit{typedec} & ::= & \texttt{type}\ \langle\, \texttt{(}\textit{tyvar}_1 \ldots \textit{tyvar}_n\texttt{)}\,\rangle\ \textit{tycon}\ \texttt{=}\ \textit{conbind} \\
\textit{conbind} & ::= & \textit{cname}\ \langle \texttt{of}\ \textit{ty}_1 * \ldots * \textit{ty}_n \rangle\ \langle\ \texttt{|}\ \textit{conbind} \rangle \\
 & & \mid\ \texttt{!}\textit{cname}\ \langle\ \texttt{|}\ \textit{conbind} \rangle \\
\textit{ty} & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \textit{tyvar} \\
 & & \mid \textit{ty}\ \texttt{array} \mid \textit{tyseq}\ \textit{tycon} \mid \textit{ty}_n\ \texttt{->}\ \ldots\ \texttt{->}\ \textit{ty}_1\ \texttt{->}\ \textit{ty} \\
\textit{valdecseq} & ::= & \textit{valdec}\ \langle \textit{valdecseq} \rangle \\
\textit{valdec} & ::= & \texttt{val}\ \textit{var}\texttt{:}\ \textit{ty} \mid \texttt{val}\ \textit{fname}\texttt{:}\ \textit{ty} \\
\textit{funimpseq} & ::= & \textit{funimp}\ \langle \textit{funimpseq} \rangle \\
\textit{funimp} & ::= & \texttt{let}\ \langle \texttt{rec} \rangle\ \textit{fundecseq} \\
\textit{fundecseq} & ::= & \textit{fundec}\ \langle \texttt{and}\ \textit{fundecseq} \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{fundec} & ::= & \mathit{fname}\ \mathit{varseq}\ \texttt{=}\ \mathit{expr} \\
\mathit{expr} & ::= & \mathit{const} \mid \mathit{var} \mid \mathit{uop}\ \mathit{expr} \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid \mathit{fname}\ \mathit{expr}_1 \ldots \mathit{expr}_n \\
& & \mid \mathit{cname}\ \texttt{(}\mathit{expr}_1,\ldots,\mathit{expr}_n\texttt{)} \mid \mathit{cname}\ \texttt{(}\mathit{expr}_1,\ldots,\mathit{expr}_n\texttt{)@}\mathit{var} \\
& & \mid \texttt{let}\ \mathit{pat}\ \texttt{=}\ \mathit{expr}\ \texttt{in}\ \mathit{expr} \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
& & \mid \texttt{match}\ \mathit{expr}\ \texttt{with}\ \mathit{match} \mid \texttt{free}\ \mathit{var} \mid \texttt{(}\mathit{expr}\texttt{)} \mid \texttt{begin}\ \mathit{expr}\ \texttt{end} \\
\mathit{match} & ::= & \mathit{mrule}\ \langle\ \texttt{|}\ \mathit{match}\rangle \\
\mathit{mrule} & ::= & \mathit{con}\langle\texttt{(}\mathit{pat}_1,\ldots,\mathit{pat}_n\texttt{)}\rangle\ \texttt{->}\ \mathit{expr} \\
& & \mid \mathit{con}\langle\texttt{(}\mathit{pat}_1,\ldots,\mathit{pat}_n\texttt{)}\rangle\texttt{@}\mathit{pat}\ \texttt{->}\ \mathit{expr} \\
\mathit{pat} & ::= & \mathit{var} \mid \texttt{\_} \\
\mathit{uop} & ::= & \texttt{-} \mid \texttt{-.} \mid \texttt{not} \\
\mathit{op} & ::= & \mathit{arithop} \mid \mathit{cmp} \mid \texttt{\^{}} \mid \texttt{\&\&} \mid \texttt{||} \\
\mathit{arithop} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{mod} \mid \texttt{+.} \mid \texttt{-.} \mid \texttt{*.} \mid \texttt{/.} \\
\mathit{cmp} & ::= & \texttt{=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{=.} \mid \texttt{<.} \mid \texttt{<=.} \mid \texttt{>=.} \mid \texttt{>.}
\end{array}
$$

There are a number of built-in operators: the operators `+`,`-`, ... `>` apply to integer values, whereas `+.`,`-.`, ... apply to floating-point values. The boolean expression e_1 `&&` e_2 is an abbreviation for `if` e_1 `then` e_2 `else false`; similarly e_1 `||` e_2 represents `if` e_1 `then true else` e_2. The remaining binary operator is `^`, which performs string concatenation. There are also three unary negation operators.

In addition there are a number of predefined functions such as `print_int`, `print_int_newline`, and `int_of_float`, whose names should explain themselves. The `same_string` function is used to compare strings for equality. There are functions for handling arrays, but we will not use these here. Camelot also includes a built-in polymorphic `list` type. In order to execute a program the user must include a function `start: string list -> unit`; when the class file is executed the `start` function will be executed with an argument consisting of a list containing the command-line arguments to the program.

Note that in some contexts the symbol `_` can be used instead of a variable name. This feature can be used to discard unwanted values such as unit values returned by print statements.

1.2.1 Basic Features of Camelot

The core of Camelot is a standard polymorphic ML-type functional language. One can define datatypes in the normal way:

```
type intlist = Nil | Cons of int * intlist
type 'a polylist = NIL | CONS of 'a * 'a polylist
type ('a, 'b) pair = Pair of 'a *'b
```

To simplify the compilation process we prohibit the `unit` type in datatype definitions. This does not cause any loss of generality since the excluded datatypes are isomorphic to types of the kind which we do allow. Values belonging to user-defined types are created by applying constructors and are deconstructed using the `match` statement:

```
let length l = match l with
       Nil -> 0
     | Cons (h,t) -> 1+length t

let test () = let l = Cons(2, Cons(7,Nil))
    in length l
```

The form of the match statement is much more restricted than in SML or O'Caml. There must be exactly one rule for each constructor in the associated datatype, and each rule binds the values contained in the constructor to variables (or discards them by using the pseudo-variable _). Complex patterns are not available, and must be simulated with further `match` and `if` statements.

As can be seen from the example above, constructor arguments are enclosed in parentheses and are separated by commas. In contrast, function definitions and applications which require multiple arguments are written in a "curried" style:

```
let add a b = a+b
let f x y z = add x (add y z)
```

Despite this notation, the present version of Camelot does *not* support higher-order functions; any application of a function must involve exactly the same number of arguments as are specified in the definition of the function.

1.2.2 Diamonds and Resource Control

Our current implementation of Camelot targets the Java Virtual Machine, and values from user-defined datatypes are represented by heap-allocated objects from a certain JVM class. Details of this representation will be given in §1.4.1.

Consider the following function which uses an accumulator to reverse a list of integers (as defined by the `intlist` type above).

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

This function allocates an amount of memory equal to the amount occupied by the input list. If no further reference is made to the input list then the heap space which it occupies may eventually be reclaimed by the JVM garbage collector.

In order to allow more precise control of heap usage, Camelot includes constructs allowing re-use of heap cells. There is a special type known as the *diamond type* (denoted by <>) whose values represent blocks of heap-allocated memory, and Camelot allows explicit manipulation of diamond objects. This is achieved by equipping constructors and match rules with special annotations referring to diamond values. Here is the `reverse` function rewritten using diamonds so that it performs in-place reversal:

```
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 first occurrence of Cons tells the compiler that the diamond value d is to be bound to a reference to the space used by the list cell. The annotation on the second occurrence of Cons specifies that the list cell Cons(h,acc) should be constructed in the diamond object referred to by d, and no new space should be allocated on the heap.

One might not always wish to re-use a diamond value immediately. This can sometimes cause difficulty since such diamonds might then have to be returned as part of a function result so that they can be recycled by other parts of the program. For example, the alert reader may have noticed that the list reversal function above does not in fact reverse lists entirely in place. When the user calls reverse, the invocation of the Nil constructor in the call to rev will cause a new list cell to be allocated. Also, the Nil value at the end of the input list occupies a diamond, and this is simply discarded in the second line of the rev function (and will be subject to garbage collection if there are no other references to it). The overall effect is that we create a new diamond before calling the rev function and are left with an extra diamond after the call had completed. We could recover the extra diamond by making the reverse function return a pair consisting of the reversed list and the spare diamond, but this is rather clumsy and programs quickly become very complex when using this kind of technique. To avoid this kind of problem, unwanted diamonds can be stored on a *freelist* for later use. This is done by using the annotation “@_” as in the following example which returns the sum of the entries in an integer list, destroying the list in the process:

```
let sum l acc = match l with
    Nil@_ -> acc
  | Cons (h,t)@_ -> sum t (acc+h)
```

The question now is how the user retrieves a diamond from the freelist. In fact, this happens automatically during constructor invocation. If a program uses an undecorated constructor such as Nil or Cons(4,Nil) then if the freelist is empty the JVM new instruction is used to allocate memory for a new diamond object on the heap; otherwise, a diamond is removed from the head of the freelist and is used to construct the value. It may occasionally be useful to explicitly return a diamond to the freelist and an operator free: <> -> unit is provided for this purpose.

There is one final notational refinement. The in-place list reversal function above is still not entirely satisfactory since the Nil value carries no data but is nonetheless allocated on the heap. We can overcome this by redefining the intlist type as

```
type intlist = !Nil | Cons of int * intlist
```

The exclamation mark directs the compiler to represent the Nil constructor by the JVM null reference. With the new definition of intlist the original list-reversal function performs true in-place reversal. No heap space is consumed or destroyed when the reverse function is applied. The ! annotation can be used for a single zero-argument constructor in any datatype definition. In addition, if every constructor for a particular datatype is nullary then they may all be preceded by !, in which case they will be represented by integer values at runtime. We have deliberately chosen to expose this choice to the programmer (rather than allowing the compiler to automatically choose the most efficient representation) in keeping with our policy of not allowing the compiler to perform optimisations which have unexpected results on resource consumption.

The features described above are very powerful and can lead to many kinds of program error. For example, if one applied the `reverse` function to a sublist of some larger list then the small list would be reversed properly, but the larger list could become partially reversed. Perhaps worse, a diamond object might be used in several different data structures of different types simultaneously. Thus a list cell might also be used as a tree node, and any modification of one structure might lead to modifications of the other. The simplest way of preventing this kind of problem is to require linear usage of heap-allocated objects, which means that variables bound to such objects may be used at most once after they are bound. Details of this approach can be found in Hofmann's paper [6]. Strict linearity would require one to write the list length function as something like

```
let length l = match l with
      Nil -> Pair (0, Nil)
    | Cons(h,t)@d ->
      let p = length t
      in match p with
           Pair(n, t1)@d1 -> Pair(n+1, Cons(h,t1)@d)@d1
```

It is necessary to return a new copy of the list since it is illegal to refer to `l` after calling `length l`.

Our compiler has a switch to enforce linearity, but the example demonstrates that the restrictive nature of linear typing can lead to unnecessary complications. Aspinall and Hofmann [1] give a type system which relaxes the linearity condition while still allowing safe in-place updates, and Michal Konečný generalises this still further in [9, 10]. As part of the MRG project, Konečný has implemented a typechecker for a variant of the type system of [9] adapted to Camelot.

A different approach to providing heap-usage guarantees is given by Hofmann and Jost in [7], where an algorithm is presented which can be used to statically infer heap-usage bounds for functional programs of a suitable form. In collaboration with the MRG project, Steffen Jost has implemented a variant of this inference algorithm for Camelot: the implementation is described in [8].

Both of these implementations are currently stand-alone programs, but we are in the process of integrating them with the Camelot compiler.

One of our goals in the design of Camelot was to define a language which could be used as a testbed for different methods of analysing heap usage. The inclusion of explicit diamonds fits the type systems of [1, 9, 10], and the inclusion of the freelist facilitates the Hofmann-Jost inference algorithm, which requires that all memory management takes place via a freelist. We believe that the fact that implementations of two radically different systems have been based on Camelot indicates that our goal was achieved successfully.

1.3 GRAIL

Instead of translating directly to JVM bytecode, the Camelot compiler targets the intermediate language Grail (Guaranteed resource allocation intermediate language). This is a small typed language which allows us to represent (a subset of) JVM bytecode in a functional form (see [13] or [23] for more information about the Java Virtual Machine and JVM bytecode). The design of Grail was inspired by the λJVM language of [11] We will give a brief overview of Grail here. For further details see [14] or [3].

A Grail program defines a single Java class, potentially containing static fields, instance fields, static methods and instance methods. Field definitions are straightforward. The real interest of Grail lies in method definitions, which are represented in a functional form whose syntax is given below.

methoddef	::=	$\mathtt{method}\ \mathit{modifiers}\ \mathit{rty}\ \mathit{jname}\ (\langle \mathit{ty}_1\ \mathit{var}_1, \cdots, \mathit{ty}_n\ \mathit{var}_n \rangle) = \mathit{methodbody}$
methodbody	::=	$\mathtt{let}\ \langle \mathit{valdec}_1 \cdots \mathit{valdec}_m \rangle\ \langle \mathit{fundec}_1 \cdots \mathit{fundec}_n \rangle\ \mathtt{in}\ \mathit{result}\ \mathtt{end}$
valdec	::=	$\mathtt{val}\ \mathit{var} = \mathit{primop} \mid \mathtt{val}\ () = \mathit{primop}$
fundec	::=	$\mathtt{fun}\ \mathit{fname}(\langle \mathit{ty}_1\ \mathit{var}_1, \cdots, \mathit{ty}_n\ \mathit{var}_n \rangle) = \mathit{funbody}$
funbody	::=	$\mathit{result} \mid \mathtt{let}\ \langle \mathit{valdec}_1 \cdots \mathit{valdec}_n \rangle\ \mathtt{in}\ \mathit{result}\ \mathtt{end}$
result	::=	$\mathit{primres} \mid \mathtt{if}\ \mathit{value}\ \mathit{test}\ \mathit{value}\ \mathtt{then}\ \mathit{primres}\ \mathtt{else}\ \mathit{primres}$
primres	::=	$\mathit{primop} \mid () \mid \mathit{fname}(\langle \mathit{var}_1, \cdots, \mathit{var}_n \rangle)$
primop	::=	$\mathit{value} \mid \mathit{binop}\ \mathit{value}\ \mathit{value} \mid \mathtt{new}\ <\mathit{condesc}>\ (\langle \mathit{value}_1, \cdots, \mathit{value}_n \rangle)$
		$\mid \mathtt{invokevirtual}\ \mathit{var}\ <\mathit{methoddesc}>\ (\langle \mathit{value}_1, \cdots, \mathit{value}_n \rangle)$
		$\mid \mathtt{invokestatic}\ <\mathit{methoddesc}>\ (\langle \mathit{value}_1, \cdots, \mathit{value}_n \rangle)$
		$\mid \mathtt{invokespecial}\ \mathit{var}\ <\mathit{methoddesc}>\ (\langle \mathit{value}_1, \cdots, \mathit{value}_n \rangle)$
		$\mid \mathtt{getfield}\ \mathit{var}\ <\mathit{fielddesc}> \mid \mathtt{putfield}\ \mathit{var}\ <\mathit{fielddesc}>\ \mathit{value}$
		$\mid \mathtt{getstatic}\ <\mathit{fielddesc}> \mid \mathtt{putstatic}\ <\mathit{fielddesc}>\ \mathit{value}$
		$\mid \mathtt{checkcast}\ \mathit{longjname}\ \mathit{var} \mid \mathtt{instanceof}\ \mathit{longjname}\ \mathit{var}$
		$\mid \mathtt{itof}\ \mathit{value} \mid \mathtt{ftoi}\ \mathit{value} \mid \mathit{arrayop}$
arrayop	::=	$\mathtt{empty}\ \mathit{value}\ \mathit{ty} \mid \mathtt{length}\ \mathit{var} \mid \mathtt{get}\ \mathit{var}\ \mathit{value} \mid \mathtt{set}\ \mathit{var}\ \mathit{value}\ \mathit{value}$
condesc	::=	$\mathit{longjname}\ (\langle \mathit{ty}_1, \cdots, \mathit{ty}_n \rangle)$
methoddesc	::=	$\mathit{rty}\ \mathit{longjname}\ (\langle \mathit{ty}_1, \cdots, \mathit{ty}_n \rangle)$
fielddesc	::=	$\mathit{ty}\ \mathit{longjname}$
test	::=	$= \mid <> \mid < \mid <= \mid > \mid >=$
binop	::=	$\mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul} \mid \mathtt{div} \mid \mathtt{mod}$
value	::=	$\mathit{var} \mid \mathit{intvalue} \mid \mathit{floatvalue} \mid \mathit{stringvalue} \mid \mathtt{null}[\mathit{longjname}]$
ty	::=	$\mathtt{int} \mid \mathtt{float} \mid \mathtt{string} \mid \mathit{longjname} \mid \mathit{ty}[]$
rty	::=	$\mathit{ty} \mid \mathtt{void}$
modifiers	::=	$\langle \mathtt{public} \mid \mathtt{protected} \mid \mathtt{private} \rangle\ \langle \mathtt{static} \rangle\ \langle \mathtt{final} \rangle$

The terms *longjname* and *jname* refer to Java-style class, field and method names; items of type *longjname* may contain dots, whereas those of type *jname* may not. In addition, Java method names for initialisers may end with `.<init>` or `.<clinit>`. The terms *var* and *fname* denote local variable names and function names respectively. Expressions of the form `val () = ...` are used to invoke operations such as `putfield` which do not return a result, and also to call `void` methods.

As a simple example of Grail, the following code defines a class containing a method for calculating the factorial of an integer.

```
class Fac {
   method public static int fac (int n) =
   let
      val b = 1
      fun f(int n, int b) = if n < 1 then b else f_else(n,b)
      fun f_else(int n, int b) =
      let val b = mul b n
          val n = sub n 1
      in f(n,b) end
   in f(n,b) end
}
```

1.3.1 The Grail Type System

Grail implements a type system similar to a subset of the JVM type system. The `int` and `float` types are the same as corresponding JVM types. There is also a collection of *reference types* which represent Java class instances and arrays. These can be used to access any Java class or method from Grail. The concrete syntax also includes a `string` type which is the same as `java.lang.String`. One major difference between the Grail and JVM type systems is that there is no subtyping in Grail. The JVM allows an object x from a class C to be used in any context where an object from a superclass S of C is expected, but Grail requires object types to match exactly. The object x must be explicitly upcast to S using the `checkcast` operation before the assignment takes place. This causes unnecessary casting operations to occur in the corresponding bytecode, but enables considerable simplifications in typechecking for Grail; furthermore, the Camelot compiler does not make any use of the Java inheritance features at present, so this point does not cause any problems.

1.3.2 Compilation of Grail

We will describe some features of the Grail compilation process. Full details can be found in [14].

In Grail, named variables are in one-to-one correspondence with JVM local variables. The JVM operand stack is used in a very restricted way in that intermediate results may not be left on the stack for later use: they must immediately be stored in a variable, leaving the stack empty. Thus to add three integers one must add the first two and store the result in some intermediate variable x say, and then add the final variable to x.

The primitive operations (the class *primop* above) correspond directly to atomic JVM operations and can be translated more or less verbatim (except for the Grail `new` operation, which combines object creation and initialisation).

Each Grail method is compiled into a JVM method. The JVM is an imperative machine with branches and `goto` statements, but these instructions are not visible in Grail. Instead, flow control within a Grail method is handled by calls to local functions defined in the method. These function calls are very restricted: they may only occur in tail position, and we require that whenever a function is called the names of its actual parameters must exactly match those used in its declaration. This convention allows a very simple translation to JVM bytecode. Function bodies are translated into basic blocks of bytecode, and every function call may assume that its arguments are already stored in the correct registers, so that the call can be translated into a direct jump.

The structure of Grail (in particular the calling convention) means that there is a very close correspondence between functional Grail and the imperative bytecode obtained by compiling it. In fact, the resulting bytecode is so idiomatic that it is easy to translate it back to the original Grail source, which is a useful feature from the PCC viewpoint. In addition, the transparency of the correspondence is important from the point of view of resource accounting: for example the calling convention means that no extra code which might affect execution time or stack size has to be introduced to place arguments in the correct registers.

The restricted form of Grail bytecode also has interesting implications for the JVM verification process. For example, the structure of the language in fact guarantees that valid Grail will compile to verifiable bytecode (we do not have a formal proof of this, but we are confident that it is true); this means that we have a *syntactic* guarantee of verifiability, whereas the verifiability of arbitrary bytecode can only be established *algorithmically*, by actually running the verification algorithm.

It also turns out that bytecode obtained from Grail is much easier to verify than arbitrary bytecode. For example, one of the conditions that bytecode must satisfy during verification is that at any particular point in a program the number and types of the elements on the operand stack is independent of the path taken to reach that point. To establish this requires an iterative process to calculate fixpoints for stack types (see [13, §4.9.2]), which can consume a lot of time and space (see [12, §2.3] for some concrete figures). In [12] Leroy examines JVM bytecode verification in detail and shows that if some simple restrictions are imposed on the form of the bytecode (notably that the stack be empty at each jump destination) then checking this property is considerably simpler. In fact, Leroy shows that the entire verification process can be carried out in linear time and constant space. The improvement is such that bytecode verification can be performed even with the extremely limited resources of a smartcard. This has hitherto been infeasible, and the standard approach has been for a trusted agent to perform off-card verification of bytecode prior to downloading. It is easily seen that Grail satisfies Leroy's conditions, which is encouraging since we hope to use it with devices with limited resources.

Some other properties of Grail are studied in [3]: among other things it is shown that the structure of Grail has connections with the well-known static single-assignment form.

We have implemented programs called `gdf` and `gf` which perform the translation from Grail to JVM and back. These can be downloaded from [15].

1.4 COMPILING CAMELOT TO GRAIL

We have implemented a Camelot compiler (available from [15]) which operates by translating Camelot into Grail and then into JVM bytecode. The compiler is a whole-program compiler whose back end is essentially the `gdf` program mentioned above. This section will describe the translation from Camelot to Grail.

1.4.1 Representing Data

Our compilation strategy is *type-preserving* in that well-typed Camelot programs are translated into well-typed Grail programs. This increases the robustness of the compiler since implementation errors often lead to type errors in the Grail code which are then detected by the Grail typechecker in the back end of the compiler.

The basic types `bool`, `int`, `float` and `string` are represented by the obvious Grail types. The `unit` type causes difficulties since there is no corresponding type in Grail. It is in fact possible to "compile away" occurrences of the unit type: this is described in an extended version of this paper available from [15].

Objects belonging to user-defined datatypes are represented by members of a single JVM class which we will refer to as the *diamond class*. Objects of the diamond class contain enough fields to represent any member of *any* datatype defined in the program. Each instance X of the diamond class contains an integer tag field which identifies the constructor with which X is associated. The diamond class also contains a static field pointing to the freelist. The freelist is managed via the static methods `alloc` (which returns the diamond at the head of the freelist, or creates a new diamond by calling `new` if the freelist is empty), and `free` which places a diamond object on the freelist. The diamond class also has overloaded static methods called `make` and `fill`, one instance of each for every sequence of types appearing in a constructor. The `make` methods are used to implement ordinary constructor application; each takes an integer tag value and a sequence of argument values and calls `alloc` to obtain an instance of the diamond class, and then calls a corresponding `fill` method to fill in the appropriate fields with the tag and the arguments. The `fill` methods are also used when the programmer reuses an existing diamond to construct a datatype value.

It can be argued that this representation is inefficient in that datatype values are often represented by JVM objects which are larger than they need to be. This is true, but is difficult to avoid due to the type-safe nature of JVM memory management which prevents one from re-using the heap space occupied by a value of one type to store a value of a different type. We wish to be able to reuse heap space, but this can be impossible if objects can contain only one type of data. With the current scheme one can easily write a heapsort program which operates entirely in-place. List cells are large enough to be reused as heap nodes and this allows a heap to be built using cells obtained by destroying the input list. Once the heap has been built it can in turn be destroyed and the space reused to build the output list. In this case, the amount of memory occupied by a list cell is larger than it needs to be, but the overall amount of store required is less than would be the case if separate classes were used to contain list cells and heap nodes.

In the current context it can be claimed that it is better to have an inefficient representation about which we can give concrete guarantees than an efficient one which about we can say nothing. Most of the programs which we have written so far use a limited number of datatypes so that the overhead introduced by the monolithic representation for diamonds is not too severe. However, it is likely that for very large programs this overhead would become unacceptably large. One possibility which we have not yet explored is that it might be possible to achieve more efficient heap usage by using dataflow techniques to follow the flow of diamonds through the program and detect datatypes which are never used in an overlapping way. One could then equip a program with several smaller diamond classes which would represent such non-overlapping types.

These problems could be avoided by compiling to some platform other than the JVM (for example to C or to a specialised virtual machine) where compaction of heap regions would be possible. The Hofmann-Jost algorithm is still applicable in this situation, so it would still be feasible to produce resource guarantees. However, it was a fundamental decision of the MRG project to use the JVM, based on the facts that the JVM is widely deployed and very well-known, and that resource usage is a genuine concern in many contexts where the JVM is used. Our present approach allows us to produce concrete guarantees at the cost of some overhead; we hope that at a later stage a more sophisticated approach (such as the one suggested above) might allow us to reduce the overheads while still obtaining guaranteed resource bounds.

1.4.2 Compilation of Programs

We compile a Camelot program to a single class with one static method for each function in the program. This technique is somewhat problematic since recursive function calls translate to recursive calls on JVM methods, which are expensive and can potentially lead to overflow of the JVM stack. Functions which call themselves in a tail-recursive manner can safely be compiled into recursive Grail function calls, and a compiler option is available which enables this feature (see [24], which also includes a proof that the optimisation has no effect on heap usage). However, mutually tail-recursive functions are difficult to program within a single stack frame because JVM methods can only have one entry point and there is a limit on the size of method bodies. Various techniques are known which can overcome this problem (for example, the *trampoline* [22, §6.2], Baker's "Cheney on the MTA" technique [2]). Unfortunately, all of these strategies tend to require extra heap usage and thus compromise the transparency of the compilation process. Because of this, at present we simply compile each function as a separate method and implement (non-recursive) tail calls as standard method calls, which carries a risk of stack overflow in programs which make a lot of use of mutual recursion. We will return to this problem in our closing remarks.

1.4.3 Initial Transformations

Compilation begins with a phase in which several transformations are applied to the abstract syntax tree.

Monomorphisation

Firstly, all polymorphism is removed from the program. For polymorphic types $(\alpha_n, \ldots, \alpha_1)\ t$ such as α `list` we examine the entire program to determine all instantiations of the type variables, and compile a separate datatype for each distinct instantiation. Similarly, whenever a polymorphic function is defined the program is examined to find all uses of the function and a monomorphic function of the appropriate type is generated for each distinct instantiation of types.

Normalisation

After monomorphisation there is a phase referred to as *normalisation* which transforms the Camelot program into a form (*Normalised Camelot*) which closely resembles Grail.

Firstly the compiler ensures that all variables have unique names. Any duplications are resolved by generating new names. This allows us to map Camelot variable names directly onto Grail variable names (which in turn map onto JVM local variable locations) with no danger of clashes arising.

We next have to simplify boolean expressions. Grail has no direct equivalent for expressions such as $m < n$ outside `if`-expressions and we deal with this by replacing such expressions with ones of the form `if` $m < n$ `then true else false`.

Next, we give names to intermediate results in many contexts by replacing complex expressions with variables. For example, the expression $f(a+b+c)$ would be replaced by an expression of the form `let` $t_1 = a+b$ `in let` $t_2 = t_1 + c$ `in` $f\ t_2$. The introduction of names for intermediate results can produce a large number of Grail (and hence JVM) variables. After the source code has been compiled to Grail the number of local variables is minimised by applying a standard register allocation algorithm (see [24]).

A final transformation ensures that `let`-expressions are in a "straight-line" form. After all of these transformations have been performed expressions have been reduced to the following form:

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{expr}' \mid \texttt{let}\ \mathit{pat} = \mathit{expr}'\ \texttt{in}\ \mathit{expr} \\
\mathit{expr}' & ::= & \mathit{primexp} \mid \texttt{if}\ \mathit{atom}\ \mathit{cmp}\ \mathit{atom}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & & \mid \texttt{if}\ \mathit{atom}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \mid \texttt{match}\ \mathit{var}\ \texttt{with}\ \mathit{match}\ \texttt{end}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{primexp} & ::= & \mathit{atom} \mid \mathit{uop}\ \mathit{atom} \mid \mathit{atom}\ \mathit{arithop}\ \mathit{atom} \mid \mathtt{free}\ \mathit{var} \\
 & & \mid \mathit{fname}\ \mathit{atom}_1 \ldots \mathit{atom}_n \mid \mathit{cname}\ \mathtt{(}\mathit{atom}_1, \ldots, \mathit{atom}_n\mathtt{)}\ \langle \mathtt{@}\mathit{var} \rangle \\
\mathit{atom} & ::= & \mathit{const} \mid \mathit{var}
\end{array}
$$

(undefined syntactic classes remain the same as those in the full syntax of Camelot given earlier). The structure of normalised Camelot (which is in fact in a type of A-normal form [5]) is sufficiently close to that of Grail that it is fairly easy to translate from the former to the latter. Another benefit of normalisation is that it is easier to write and implement type systems for normalised Camelot. The fact that the components of many expressions are atoms rather than complex subexpressions means that typing rules can have very simple premisses.

1.4.4 Compilation of expressions

The Camelot expressions labelled by the term *primexp* in the normalised syntax above will be referred to as *primitive expressions*. They are significant because they correspond directly to primitive operations in Grail, and thus admit an easy translation. This is the key to compilation of normalised Camelot into Grail. A normalised Camelot expression consists of a nested sequence of `let` expressions. The translation procedure essentially translates an expression (in particular, a function body) into a collection of mutually recursive Grail local functions by descending down the chain of `let`-expressions, emitting a Grail *valdec* for each term of the form `let` $p = e$ with e primitive. This process terminates when a non-primitive expression e is encountered; at this point e must be a branch of some kind, and the compiler recursively generates a new local function for each of the subexpressions occurring in the branch, terminating the original function with a Grail `if`-result (or, in the case of a `match` statement, a block of code implementing a sequence of such results). This a highly simplified description of the translation to Grail; space constraints preclude a full description, but the extended version of this paper (see [15]) contains an appendix giving a full and precise specification of the translation.

1.5 PERFORMANCE

We have described a procedure for compiling Camelot into Grail, and thence to JVM. This is a long process involving several different stages, and one might suspect that it would introduce inefficiencies into the final bytecode programs. In this section we will present figures comparing the run-time of various Camelot programs with versions of the same programs written in Java and in Scheme, which we hope will demonstrate that performance is not compromised unduly.

Java programs were compiled using the standard Sun Java compiler. To compile Scheme programs for the JVM the Bigloo Scheme compiler [20, 19] was used.

Timings were obtained using the JFluid JVM profiling tool [4]; this uses a special version of the Sun JVM (version 1.4.2) which has been modified to allow dynamic instrumentation of class files. The figures which are obtained appear to be fairly accurate since one can focus on particular areas of the program without incurring an overhead by profiling irrelevant code. By default the JVM performs adaptive compilation to native code for frequently-executed code sequences. This feature is not available in JFluid, so all execution was performed by interpretation. However, we felt that this would still give a realistic (worst-case) estimate of program times. Also, JVMs for limited-memory devices generally provide no alternatives to interpretation. The timings were carried out on a 366MHz Pentium 2 processor under Linux. All timings are in milliseconds and represent an average taken over five runs.

Firstly we consider several list-reversal programs. Each program generates a list consisting of the integers between 1 and 1,000,000 and then proceeds as follows:

- A reverses the list in place.
- B reverses the list in place, but replaces each element x by $x+x$.
- C returns a reversed copy of the list, leaving the original intact.
- D returns a reversed copy with each element doubled as in B.

We timed the execution of the entire program (including construction of the input list), and also of the reversal function in isolation. The results follow below.

	A		B	
	main	reverse	main	reverse
Java	6289ms	507ms	6653ms	850ms
Camelot	11263ms	1684ms	11684ms	1785ms
Scheme	28884ms	3645ms	58595ms	30734ms

	C		D	
	main	reverse	main	reverse
Java	10824ms	5009ms	10670ms	5215ms
Camelot	20285ms	10439ms	20580ms	10676ms
Scheme	31686ms	6829ms	54178ms	28822ms

We note that the Camelot versions are slower than the Java versions, but are generally faster than the Scheme versions. There are several reasons why Camelot is slower than Java.

(1) The requirement that all intermediate results in Grail are explicitly named means that the bytecode often contains pairs of instructions where a value is stored in a local variable and then immediately recalled for further use (and the stored copy is never used again). This certainly has the effect of slowing down the execution of the bytecode, but the decision to use this form of code was made deliberately in the hope that the regularity of the bytecode would simplify formal analyses.

(2) In Camelot it is not possible to modify individual fields within an object: when a value is constructed in a recycled Camelot diamond, the fields in the corresponding object are filled in by a method call (to the `fill` method mentioned in 1.4.1). All fields must be explicitly rewritten, even if some have not changed (see the reversal example in 1.2.2, which is essentially the same as the one used in program A). In contrast, in Java one can perform list-reversal simply by changing pointers in list cells and leaving the other values stored in the cells intact. This accounts for the fact that simple in-place reversal in Java is three times as fast as in Camelot, but when the entries in the list are modified, as in program B, the Java version is only twice as fast as the Camelot version. The fact that a method call is used, rather than a sequence of `putfield` operations, also adds some extra overhead. Again, this was a conscious design decision: a constructor application in Camelot corresponds directly to a single method application, and it was felt that this correspondence would simplify analysis.

We performed the Scheme comparisons as we thought it would be interesting to compare Camelot's performance with that of another functional language running on the JVM. It was somewhat surprising to discover that while Scheme took six times as long as Java to perform simple in-place list reversal, it took more than 36 times as long to perform reversal with doubling. This appears to be due to the fact that Scheme's numeric `+` operator is overloaded. Inspection of the bytecode produced by the compiler reveals that Bigloo handles overloading by representing numeric values in a boxed form as Java objects. When elements in the list are doubled, this requires the `+` operator to examine the boxed values to determine their numeric type and then call an appropriate specialised addition operator, and then to re-box the result prior to insertion in the modified list. Since this happens for each of the million elements in the list it is not surprising that there is a considerable slowdown. By using the Scheme `+fx` operator in place of `+` it is possible to use Scheme `fixnum` values, which Bigloo encodes as JVM `int` values. When program B is modified in this way the execution time for the reversal function reduces to about 14000ms. This figure is still about 16 times as long as the Java version: we suspect that this is largely due to the fact that dynamic typechecking is still required before the addition operator is actually called.

The following table gives timings for some other programs:

	Fibonacci	Quicksort	Insertion Sort
Java	221229ms	21009ms	23963ms
Camelot	239039ms	34166ms	42415ms
Scheme	709598ms	42368ms	73412ms

The first column gives times for calculation of the 40th Fibonacci number by a direct implementation of the recursive definition. Execution of the program consists mostly of recursive method invocations, so the performance of Java and Camelot is very similar. Again Scheme performs badly due to dynamic typechecking. The figures given represent a calculation using `finxum` values; when these were replaced by the default boxed integer values, the execution time rose to 6577734ms, or about 1 hour and 49 minutes.

The second column of the table gives times for execution of an in-place quicksort algorithm on a list of 25586 words (the text of [21]), and the third column gives times for an in-place insertion sort of a list consisting of the first 5000 words of the same list. Again Java performs best, with Camelot second and Scheme third, but in these examples the differences are less marked than in some of the previous examples.

Overall the figures show that Camelot programs compare favourably with Java programs. Furthermore, it is fairly clear which features of Camelot are responsible for its poorer performance. As suggested above, the somewhat rigid structure of the bytecode obtained from Camelot programs is due to deliberate design decisions which were made in order to allow a precisely-defined and transparent compilation procedure which would facilitate program analysis. It is possible that some of these restrictions could eventually be relaxed (thereby improving performance) without compromising the validity of our analyses.

We have only considered execution time here. Of course, our main interest is in memory usage. JFluid also allows one to collect memory profiling information, and this indicates that the heap usage of the Java and Camelot programs was exactly as expected. Unfortunately we were unable to obtain any heap profiling for the Scheme programs since they appeared to terminate in a nonstandard way which the JFluid system was unable to deal with properly.

1.6 FINAL REMARKS

We have described a technique for compiling Camelot into JVM bytecode via the functional intermediate language Grail; we believe that this technique satisfies the strict requirements of the PCC framework. We have also provided some performance figures which indicate that the rigid specification of the compilation procedure does not degrade execution speed unduly.

There are various ways in which Camelot could be extended. The lack of higher-order functions is inconvenient, but the resource-aware type systems which we use are presently unable to deal with higher-order functions, partly because of the fact that these are normally implemented using heap-allocated closures whose size may be difficult to predict. A possible strategy for dealing with this which we are currently investigating is Reynolds' technique of *defunctionalization* [18] which transforms higher-order programs into first-order ones, essentially by performing a transformation of the source code which replaces closures with members of datatypes. This has the advantage that extra space required by closures is exposed at the source level, where it is amenable to analysis by the heap-usage inference techniques mentioned earlier.

A similar strategy can be used to eliminate mutual tail-recursion. Given a set of mutually recursive functions $\mathcal{F}$ whose results are of type `t`, we define a datatype `s` which has for each of the functions in $\mathcal{F}$ a constructor with arguments corresponding to the function's arguments. The collection of functions $\mathcal{F}$ is then replaced by a single function `f: s -> t` whose body is a `match` statement which carries out the computations required by the individual functions in

$\mathcal{F}$. In this way the mutually recursive functions can be replaced by a single tail-recursive function, and we already have an optimisation which eliminates recursion for such functions. This technique is somewhat clumsy, and care is required in recycling the diamonds which are required to contain members of the datatypes required by `s`. Another potential problem is that several small functions are effectively combined into one large one, and there is thus a danger that that 64k limit for JVM methods might be exceeded. Nevertheless, this technique does overcome the problems related to mutual recursion without affecting the transparency of the compilation process unduly, and it might be possible for the compiler to perform the appropriate transformations automatically. We intend to investigate this in more detail.

As an extension in a different direction, the second author has recently extended the language (and the compiler) to include object-oriented features and allow the use of pre-existing Java libraries: details can be found in [25].

As mentioned earlier, complex resource-aware type-systems and inference methods have been implemented for Camelot and will soon be integrated with the present compiler. Eventually, the MRG project aims to have a certifying compiler which will take a Camelot program and automatically provide a proof that it abides by a given resource policy.

Acknowledgments

The authors would like to thank Hans-Wolfgang Loidl and Ian Stark for their comments.

REFERENCES

[1] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proc. 11th European Symposium on Programming, Grenoble*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.

[2] Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–20, September 1995.

[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] M. Dmitriev. Welcome to JFluid, October 2003. See http://research.sun.com/projects/jfluid.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

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

[7] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages, New Orleans*, 2003.

[8] S. Jost. `lfd_infer`: an implementation of a static inference on heap-space usage. In *Proceedings of SPACE'04, Venice*, 2004. To appear.

[9] Michal Konečný. Functional in-place update with layered datatype sharing. In *TLCA 2003, Valencia, Spain, Proceedings*, pages 195–210. Springer-Verlag, 2003. Lecture Notes in Computer Science 2701.

[10] Michal Konečný. Typing with conditions and guarantees for functional in-place update. In *TYPES 2002 Workshop, Nijmegen, Proceedings*, pages 182–199. Springer-Verlag, 2003. Lecture Notes in Computer Science 2646.

[11] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[12] Xavier Leroy. Bytecode verification for Java smart cards. *Software Practice & Experience*, 32:319–340, 2002.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999. Available at http://java.sun.com/docs/books/vmspec/.

[14] K. MacKenzie. Grail: a functional intermediate language for resource-bounded computation. LFCS, University of Edinburgh, 2002. Available at http://www.dcs.ed.ac.uk/home/mrg/publications/.

[15] The Mobile Resource Guarantees project. http://www.dcs.ed.ac.uk/home/mrg.

[16] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: 24th ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[17] O'Caml. Welcome to the O'Caml language, October 2003. See www.ocaml.org.

[18] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.

[19] M. Serrano. See http://www-sop.inria.fr/mimosa/fp/Bigloo.

[20] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[21] Robert Louis Stevenson. *Strange Case of Dr. Jekyll and Mr. Hyde*. Longmans, Green, London, 1886. Available online at http://www.gutenberg.org.

[22] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[23] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, second edition, 1999. Also at http://www.artima.com/insidejvm/blurb.html.

[24] N. Wolverson. Optimisation and resource bounds in Camelot compilation. Final-year project report, University of Edinburgh, 2003. Available at http://www.dcs.ed.ac.uk/home/mrg/publications/wolverson.ps.

[25] N. Wolverson and K. MacKenzie. O'Camelot: adding objects to a resource-aware functional language. In *Proceedings of TFP2003*, pages ??–?? Intellect, 2004.