Grail: a functional intermediate language for resource-bounded computation. Version 1.2*

K. MacKenzie

12th December 2002

1 Introduction

Grail (Guaranteed resource allocation intermediate language) is a small typed functional language which is designed to be used as an intermediate language during the compilation process from the language described in [14] (which is now referred to as Camelot) to Java bytecode. It is intended that Grail should target a subset of the Java Virtual Machine instruction set (see [8] or [15]) which is similar to that described in [11] and that the translation from Grail to JVM bytecode should be a straightforward task; furthermore, it should also be easy to convert suitably formed Java bytecode back into Grail. The Grail language represents many of the JVM instructions directly as primitive operations while hiding details of the usage of the JVM stack and JVM local variables.

Grail was inspired by the λJVM of League et. al. (see [5] and [6]), but is much simpler than λJVM. λJVM was designed for the analysis and optimisation of arbitrary Java bytecode and hence must be able to represent any well-formed Java method. Our language is designed primarily so that it will be easy to translate into and out of (a restricted subset of) Java bytecode, and hence the language can be much simpler. The language is further simplified by the omission of features of JVM bytecode such as exceptions, subroutines and multithreading.

1.1 Terminology

At an earlier stage in the development of these ideas the JVM subset described in [11] was called "Grail" and the functional language was called "λGrail". It has now been decided that the functional language should be referred to simply as "Grail"; if required the JVM subset of [11] may be referred to as "jGrail". In

*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.

[10] we describe a subset of the .NET instruction set which corresponds closely to the set of jGrail instructions: this set of .NET instructions is referred to as .Grail.

2 Grail

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. The body of a Grail method definition essentially consists of a sequence of *value definitions* which associate values with variables, followed by a sequence of *local function definitions* and then an expression giving the method result. A local function definition consists of a sequence of value definitions followed by a *result* which can either be a *primitive result* or an *if* statement choosing between two primitive results. A primitive result is either a *primitive operation* (such as returning the contents of a variable) or a call to another local function. This is the only context in which the invocation of a local function is permitted. This means that all local function calls occur in tail position. We also enforce a very rigid calling convention for local functions (see below) which allows us to compile functions as basic blocks of JVM bytecode with function invocation being implemented as a simple jump.

To clarify this we give a formal definition of the syntax of Grail. This will be followed by explanatory remarks and examples; we will then provide a more detailed discussion of some of the features of Grail.

2.1 The syntax of Grail

The presentation of the syntax is modelled on that of SML in [12]. The symbols m and n appearing in some productions denote integers greater then zero. Angular brackets $\langle \cdots \rangle$ indicate optional items.

classdef	::=	`class` *longjname* {⟨*fielddefseq*⟩⟨*methoddefseq*⟩}
fielddefseq	::=	*fielddef* ⟨*fielddefseq*⟩
fielddef	::=	`field` *modifiers ty jname*
methodddefseq	::=	*methoddef* ⟨*methodddefseq*⟩
methoddef	::=	`method` *modifiers rty jname* (⟨ ty_1 var_1, ⋯, ty_n var_n ⟩) `=` *methodbody*
methodbody	::=	`let` ⟨$valdec_1$ ⋯ $valdec_m$⟩ ⟨$fundec_1$ ⋯ $fundec_n$⟩ `in` *result* `end`
valdec	::=	`val` *var* `=` *primop*
		`val ()` `=` *primop*
fundec	::=	`fun` *fname* (⟨ ty_1 var_1, ⋯, ty_n var_n ⟩) `=` *funbody*
funbody	::=	*result*
		`let` ⟨$valdec_1$ ⋯ $valdec_n$⟩ `in` *result* `end`
result	::=	*primres*
		`if` *value test value* `then` *primres* `else` *primres*
primres	::=	*primop*
		`()`
		fname (⟨var_1, ⋯, var_n⟩)

primop	::=	*value*
		binop value value
		`new <`*condesc*`>` `(`$\langle$*value*$_1$, $\cdots$, *value*$_n\rangle$`)`
		`invokevirtual` *var* `<`*methoddesc*`>` `(`$\langle$*value*$_1$, $\cdots$, *value*$_n\rangle$`)`
		`invokestatic <`*methoddesc*`>` `(`$\langle$*value*$_1$, $\cdots$, *value*$_n\rangle$`)`
		`invokespecial` *var* `<`*methoddesc*`>` `(`$\langle$*value*$_1$, $\cdots$, *value*$_n\rangle$`)`
		`getfield` *var* `<`*fielddesc*`>`
		`putfield` *var* `<`*fielddesc*`>` *value*
		`getstatic <`*fielddesc*`>`
		`putstatic <`*fielddesc*`>` *value*
		`checkcast` *longjname var*
		`instanceof` *longjname var*
		`itof` *value*
		`ftoi` *value*
		arrayop
arrayop	::=	`empty` *value ty*
		`length` *var*
		`get` *var value*
		`set` *var value value*
condesc	::=	*longjname* `(`$\langle$*ty*$_1$, $\cdots$, *ty*$_n\rangle$`)`
methoddesc	::=	*rty longjname* `(`$\langle$*ty*$_1$, $\cdots$, *ty*$_n\rangle$`)`
fielddesc	::=	*ty longjname*
test	::=	`=`
		`<>`
		`<`
		`<=`
		`>`
		`>=`
binop	::=	`add`
		`sub`
		`mul`
		`div`
		`mod`
value	::=	*var*
		intvalue (literal integer)
		floatvalue (literal float)
		stringvalue (literal string)
		`null[`*longjname*`]`
ty	::=	`int`
		`float`
		`string`
		longjname
		ty`[]`
rty	::=	*ty*
		`void`
modifiers	::=	$\langle$*accessmodifier*$\rangle\langle$`static`$\rangle\langle$`final`$\rangle$
accessmodifier	::=	`public`
		`protected`
		`private`

Notes on the syntax

- A Grail "program" consists of a single *classdef* (class definition), which is translated into a single JVM classfile.
- We also allow comments in either of the usual Java formats, although unlike Java or C, `/*···*/` comments nest.
- Field and method definitions may be equipped with modifiers. These are not yet permitted for class definitions. At present all classes are `public` and `final` by default.
- The phrases *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 phrases *var* and *fname* refer to local variable names and function names respectively. These are similar to ML names, and are described by the regular expression `[A-Za-z][A-Za-z0-9_]*'*`.
- 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.
- Literal strings are delimited by double quotes (`"`). Literal quotes, newlines etc. may be included using the usual backslash conventions.

2.2 Examples

The following Grail 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
}
```

The next example defines a class containing a method which will calculate the factorial of a Java `BigInteger`. The basic structure of this example is similar to the previous one.

```
class Bigfac {
   method public static
          java.lang.BigInteger fac (java.lang.BigInteger n) =
   let
      val ONE = getstatic
                  <java.lang.BigInteger java.lang.BigInteger.ONE>
      val b = ONE

      fun f (java.lang.BigInteger n,
             java.lang.BigInteger b,
             java.lang.BigInteger ONE) =
      let
         val comparison = invokevirtual n
             <int
               java.lang.BigInteger.compareTo(java.lang.BigInteger)>
                  (ONE)
      in
         if comparison < 0
            then b
            else f_else(n,b,ONE)
      end

      fun f_else (java.lang.BigInteger n,
                  java.lang.BigInteger b,
                  java.lang.BigInteger ONE) =
      let
         val b = invokevirtual b
                 <java.lang.BigInteger
                    java.lang.BigInteger.multiply
                       (java.lang.BigInteger)> (n)
         val n = invokevirtual n
                 <java.lang.BigInteger
                     java.lang.BigInteger.subtract
                        (java.lang.BigInteger)> (ONE)
      in
         f(n,b,ONE)
      end

    in
       f(n,b,ONE)
    end
}
```

2.3 The Grail type system

Grail implements a type system similar to a subset of the JVM type system. There are two *primitive types*, `int` and `float`. Values of type `int` are 32-bit signed two's-complement integers, and values of type `float` are 32-bit single precision IEEE 754 floating-point numbers. In addition to these primitive types there is 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, as in the second example above. Class names are specified as dot-separated names in the usual Java style, while array types are denoted by the `[]` symbol; for instance the type of a one-dimensional array of integers is denoted by `int[]` and the type of a two-dimensional array of strings (in fact an array of arrays of strings) is denoted by `java.lang.String[][]`. The concrete syntax also includes a `string` type: this is a synonym for `java.lang.String`. There is an implicit unit type with a single value `()` which is used to represent the return value of methods of `void` return type and the value of primitive operations such as `putfield` which return no result. We will not give explicit typing rules for Grail as the type system is straightforward.

Every variable in a Grail method has a unique type, which is inferred statically. A variable within a function f may only appear on the right-hand side of an expression if it has previously been declared within f, either as a parameter or in the left-hand side of a preceding assignment. This allows us to deduce the type of an expression solely from the constituent primitive operations and the variables appearing in the expression. We also require that within a given method a variable is only used with a single type; if some local function uses a variable x with type `int` say then no other function belonging to the same method may use x with any other type.

There is one situation in which a variable can be used without having been explicitly declared: an instance method m (*ie* one which has not been declared to be `static`) has an implicitly declared parameter called `this` representing the instance with which the method is associated. The type of the variable `this` is the same as that of the class in which m is defined.

The Grail type system is more restrictive than the Java type system. In Java a variable of reference type X can contain an object from any subclass of X. In Grail the types must match exactly. For example, if a variable x has been inferred to have type `java.lang.Object` then it is not permitted to assign a value of type `java.lang.String` to x; the value must be explicitly cast to type `java.lang.Object` using the `checkcast` operation before the assignment takes place. This causes redundant casting operations to occur in the corresponding bytecode, but enables considerable simplifications in typechecking for Grail. In addition we expect most Grail code to be generated by a compiler in a situation where complex use of the Java class hierarchy is not required. Another consequence of this restriction on the class hierarchy is that `null` reference values require type annotations to specify the class to which they refer: for instance, one must write `null[java.lang.Integer]` rather than just `null`.

Another point to note is that we do not need to specify the return types of

functions. In fact since function calls can only appear in tail position it can be shown that all functions must have the same return-type as does the method in which they are defined. This is also verified during typechecking. [In fact this condition might not hold if we allow functions which are never called; to rule this out we require that all functions are reachable from the "top" of the method. This is not yet implemented in the typechecker, and the decompiler may have problems with unreachable functions. This will be dealt with at some time in the future].

2.4 Argument passing in Grail

Function calls are very restricted in Grail. We require that whenever a function is called its list of actual parameters must exactly match those used in its declaration. If a function is declared as `fun f(int n, java.lang.String s)` then every call to `f` must have the form `f(n,s)`. This convention allows a very simple translation to JVM bytecode; every function call may assume that its arguments are already stored in the correct registers and the call can thus be translated into a direct jump. There will be further discussion of this point in Section 3.1 below.

2.5 Primitive operations

Most of the primitive operations defined above correspond directly to JVM instructions. We will describe these individually.

2.5.1 `new`

The `new` instruction is slightly different from the JVM `new` instruction. In JVM bytecode the `new` instruction simply creates a new object on the stack. The JVM verifier requires that all objects be initialised prior to their first use, but it is possible to create an object and then perform some other computations before initialising it. The Grail `new` instruction contains a descriptor specifying an initialisation method (*ie* a constructor) which is to be called immediately after the creation of the object.

2.5.2 `invokevirtual`

This instruction takes a variable name together with a descriptor specifying an instance method from some class and applies that method to a list of values (*ie* variables or literal values). For example, we might have

```
invokevirtual b <java.lang.BigInteger
                    java.lang.BigInteger.multiply
                       (java.lang.BigInteger)> (n)
```

This applies the instance method `multiply` for the object in `b` to the object in `n`.

The syntax of the `invokevirtual` instruction may appear overly complex in comparison with the corresponding Java statement `b.multiply(n)`. However, a Java compiler has to do a considerable amount of compile-time inspection of the Java class hierarchy in order to resolve overloading and overriding of methods. If we have a Java statement `x.f(a,b)` then the signature of `f` cannot be deduced simply by looking at the types of `a` and `b`. For example, if a class `C` defines static methods `f(Object a, Object b)`, `f(Object a, String b)` and `f(String a, Object b)` then a call `C.f(a,b)` might actually call any of these methods. The Java compiler has to determine which is the correct method from the compile-time types of the arguments. The situation is further complicated by the fact that it may not be possible to determine a unique method. For example, in the above situation a Java compiler must reject a call to `C.f(String, String)` since both the second and third versions of `f` are suitable. However, superclasses of `C` may also define methods called `f` which would be eligible; in the example above if `C` has a superclass `B` defining a method `B.f(String, String)` then the call to `C.f(String, String)` would in fact be acceptable.

Since Grail is intended as an intermediate language it will generally be the case that method descriptors will be automatically generated and hence their correctness can be relied upon (if not, the bytecode will be rejected by the JVM during verification). Hence we have decided to adopt the complex syntax above in order to avoid costly inspection of the Java class hierarchy.

2.5.3 `invokespecial`

This is similar to the `invokevirtual` instruction but is used for invocation of instance initialisation methods (which have the special name `<init>`).

2.5.4 `invokestatic`

This is similar to the `invokevirtual` instruction but is used to invoke static methods.

2.5.5 `getfield`, `putfield`, `getstatic`, `putstatic`

These instructions are used for getting and setting instance and static fields of classes. Hopefully the syntax requires no further explanation.

2.5.6 `checkcast`

The `checkcast` operation takes as arguments the name of a Java class (`java.lang.Integer` for instance) and the name of a Grail local variable. If the object contained in the variable can be cast to the specified class then nothing happens, otherwise `java.lang.ClassCastException` is raised by the JVM.

2.5.7 `instanceof`

This takes the name of a Java class (or interface or array type) and the name of a Grail local variable as arguments. If the variable is an instance of the class then the integer 1 is returned, otherwise 0 is returned.

2.5.8 `itof`, `ftoi`

These instructions convert values from type `int` to `float` and back.

2.5.9 Array operations

Various primitive operations are provided for manipulating arrays. All arrays are one-dimensional; multi-dimensional arrays are represented by arrays whose entries are themselves arrays. The array operations are as follows.

- `empty` n t. This creates an array of size n, all of whose elements are initialised to an appropriate zero value (zero for numeric values, `null` for objects (including strings)).
- `length` A. Return the length of the array A.
- `get` A i. Return the value of the ith entry of the array A.
- `set` A i v. Set the value of the ith entry of the array A to be v.

It should be noted that the use of arrays in Java and the JVM is not particularly efficient. To extract an element of a multidimensional array several `get` operations may be required (one for each dimension). Furthermore, array types interact badly with the Java class hierarchy. Since arrays are imperative objects they should not admit subtyping (see [13, p198]). However, the Java specification decrees that if A is a subclass of B then A`[]` should be a subclass of B`[]`. This means that whenever a `set` operation is performed the JVM must check the type of the object being inserted to make sure that it is compatible with the type of the array. This check adds an invisible overhead to array operations. Presumably the full check is unnecessary for primitive types such as `int` and `float`, but care should be exercised with arrays containing reference types.

2.5.10 Binary operations

The binary operations `add`, `sub`, `mul`, `div` and `mod` are provided, and may be applied to pairs of integers or pairs of floats. Note that floats and integers may not be mixed; the operations `itof` and `ftoi` may be used to convert between integers and floats.

2.5.11 Comparisons

The comparisons `=`, `<>`, `<`, `<=`, `>` and `>=` are provided. These can be applied to pairs of integers or pairs of floats, for which they carry out the obvious numerical comparisons. The `=` and `<>` comparisons can also be applied to pairs of objects (which must be from the same class), where they test for equality and inequality of references.

3 Some remarks on Grail

Grail has been designed to allow easy translation to and from JVM bytecode. It is hoped that the fact that Grail may be viewed as a functional language will simplify the analysis of programs. Grail can be given a semantics in a similar form to other functional languages and there is a large and well-developed body of theory which can be applied to such semantics. On the other hand, the translation process (which we will describe in more detail shortly) allows Grail to be viewed as having an imperative semantics inherited from the JVM. In [1] it is shown that, at least for a subset of Grail, these two semantics coincide: an expression evaluates to a particular result with respect to the functional semantics if and only if the expression terminates and produces the same result with respect to the imperative semantics.

There are several other reasons for using a functional intermediate language. One is that the high-level language Camelot is itself functional and it can be translated into Grail in a fairly straightforward manner. Various transformations are applied to Camelot abstract syntax to convert it into a form close to the Grail abstract syntax and the main complication in the compilation process is then to break Camelot expressions into a sequence of Grail functions where all calls (and branches) occur in tail position. See [9] for details of this process.

Another advantage of the functional form of Grail (and in particular the type system) is that it makes it easier for us to guarantee that the compiled form of a Grail program will be accepted by the JVM bytecode verifier; in fact the bytecode which we generate is so regular that the verification process is far simpler than for arbitrary JVM bytecode. In connection with this, Leroy [7] has shown that bytecode satisfying certain restrictions can be verified in linear space (usually less than 100 bytes for typical Java programs) and usually only requires two passes over the code; this allows verification to be carried out within the limited resources of a smartcard. It is easy to see that our bytecode satisfies Leroy's conditions, which are that (a) the operand stack is empty at all branch instructions and at all branch target instructions, (b) that each register has only one type throughout a method, and (c) that on entry to a method all registers (with the exception of those containing the method parameters) are zeroed. This last condition depends on the JVM and not on the bytecode, but is satisfied by most JVM implementations. Leroy uses this condition to simplify the verification of type-safety by omitting certain checks on register initialisation.

Despite the remarks above it cannot be guaranteed that an arbitrary Grail program will translate into verifiable bytecode. The main problem is that we do not inspect the Java class hierarchy during compilation in order to guarantee type-safety. We could for example emit a method call which includes a descriptor specifying a non-existent method from the Java class libraries. A Java compiler will usually check during compilation that such a method exists and will reject the program if not. As mentioned previously we have omitted these checks in order to avoid a considerable complication in the compiler and a large overhead in the compilation procedure. Since Grail is supposed to be an intermediate language we hope to be able to guarantee that the overall compilation procedure from Camelot to JVM bytecode will avoid the sorts of problems mentioned above.

3.1 Reversing the translation

We have also designed Grail so that it will be easy to translate a compiled Grail program back to Grail source. The main reason for this is that Grail programs are supposed to be *mobile*. We wish to prove statements about the resource usage of a program and then transmit the program together with a proof enabling the end-user to confirm the requirements of the program before running it. Since we aim to prove properties of Grail programs, rather than JVM code, the Grail source must be available for the end-user to examine, and this requires us to provide some form of transmission format for Grail source. It seemed that a simple solution to this problem would be to use the Java class file format itself (perhaps with some metadata specific to Grail) as the mechanism for transmitting Grail programs. This solution has the advantages that no extra work is required to define a new file format and that no extra compilation is required by the end-user. Our aim has therefore been to ensure that decompilation of Grail is easy. This explains some of the unusual features of Grail, in particular the parameter-passing mechanism. It would not be difficult to define a form of Grail in which a more normal form of parameter-passing is used and to arrange for the compiler to insert extra code to ensure that function arguments are put in the correct registers prior to a jump; however, this would mean that new code would be generated in a manner which would be difficult for the programmer to predict, and it would complicate the decompilation procedure. For this reason and also to facilitate the correspondence between functional and imperative semantics we have chosen the mechanism used here.

4 Compiling Grail

In implementations of Grail compilers and decompilers we use tools which ease the construction of JVM class files (see later for details of specific implementations). Because of this we may ignore details of how to create class definitions and set their access permissions and how to declare fields and methods. The main difficulty arises in compiling bytecode which implements method bodies,

and this is what we will concentrate on here.

4.1 Variables

Recall that each invocation of a Java method is associated with a single *stack frame* which contains an array of local variables and a finite *operand stack* which is used to store intermediate results. The operand stack is also used during method invocation. When the currently executing method wishes to invoke another method, the arguments are loaded onto the stack and then an appropriate `invokevirtual` or `invokestatic` instruction is executed.

The restrictions placed on variable usage mean that we may map each Grail variable onto a single JVM local variable with no danger of using uninitialised variables. Another consequence of the restrictions is that each JVM local variable is used with a unique type throughout the lifetime of a given method; this is one of the conditions required by Leroy in [7], and is also the convention required by .NET (see [4] and [10]).

4.2 Constants

There are several ways of loading constants onto the stack in JVM bytecode. For example, to load the integer constant 1, one could use `iconst_1` (one byte), `bipush 1` (two bytes), `sipush 1` (three bytes) or an appropriate `ldc` instruction (two bytes plus a five-byte entry in the runtime constant pool). This applies mostly to integers; all floats apart from 0.0, 1.0 and 2.0 require a `ldc` instruction. We assume that when the compiler emits instructions used to load constants then they are no larger than necessary.

4.3 Primitive Operations

Every primitive operation in Grail corresponds to a single JVM instruction (with the exception of thes `new` instruction: see above). The arguments of primitive operations can only be Grail variables or literal constants. To compile a primitive operation p therefore, we simply emit a series of instructions which load the appropriate arguments onto the stack (using `iload`, `fload` or `aload`) and then emit the instruction corresponding to p itself. After executing this instruction, the stack will contain at most one item; this can then be stored in an appropriate variable if p occurs in a statement of the form `val` x `=` p or left on the stack if p occurs as the result of some Grail function. The `new` instruction is treated similarly, except that the bytecode emitted consists of the JVM `new` instruction followed by a call to `invokespecial` to call the appropriate initialiser. We have not mentioned the field and method descriptors which are embedded in some primitive operations. These are essentially translated literally.

Binary operations are similar to other primitive operations except that our arithmetic operations are slightly more abstract than the JVM instructions in

that type information is omitted. When an `add` operation is compiled for example, the type of its arguments is checked and an `iadd` or `fadd` instruction emitted as appropriate.

4.4 Value declarations

These can take two forms:

- `val` *var* `=` *primop*
- `val () =` *primop*

The first of these requires that the primitive operation leaves a single value on the stack and the second that the primitive operation leaves the stack empty (and these requirements are enforced by the typechecker). To compile these we emit the code for performing the primitive operation; in the second case no other action is required, but in the first case we have to append an instruction which stores the result in the appropriate local variable.

4.5 Comparisons

Comparisons are only allowed in the context of `if...then...else...` statements, which may themselves only occur as the results of functions. The JVM instruction set deals with comparisons in a somewhat complex way. For integers there are tests `if_icmpeq`, `if_icmple`, ... (six tests in all) which compare two integers on the stack. If the test is positive then control branches to a specified offset; otherwise execution continues with the instruction following the test. There are also another six comparisons (`ifeq`, `ifle`, ...) which compare the item on top of the stack against 0, again branching if required. For floats, there are only two comparisons, `fcmpl` and `gcmpl`. These two instructions behave identically unless the IEEE floating-point value NaN is involved: see [8] for details. Both instructions compare two floats on the stack, leaving the integer value 1 on the stack if the value on level one of the stack is less than the value on level two, the integer 0 if the two values are equal, and -1 otherwise. A branch can then be obtained by using one of the instructions `ifeq`, `ifle`, ... mentioned above. We therefore compile `<` to `if_icmplt` for integer values and to `fcmpl` followed by `iflt` for floats. For comparisons of objects we use the `if_acmpeq` and `if_acmpne` instructions.

4.6 Function definitions

When compiling a function we create a label which we associate with the name of the function and then emit code for the body of the function. Recall that a function body consists of a sequence of value declarations followed by a *result*. To compile a function body we simply compile the value declarations sequentially, emitting a block of bytecode for each, and then add some code to compute the result. Consider first the case of a *primitive result*, which is one of: (a) a

primitive operation; (b) the unit value `()`; or (c) a function application. In case (a) we simply compile the primitive operation and then emit an appropriate return statement (`return`, `ireturn`, `freturn` or `areturn`). In case (b) we emit the `return` instruction. In case (c) we emit a jump (`goto` or `goto_w`) to the label corresponding to the name of the function to be called.

If the result of a function is not primitive then it must be an `if` statement selecting one of two possible primitive results. To deal with this we create a new label l and then produce code corresponding to the comparison in the `if` statement which jumps to the new label if the test is positive and continues without jumping if the test is negative. We then emit the code for the second primitive result, followed immediately by the code for the first primitive result (labelled by l). If the first result happens to be a function call then this process will produce a jump whose target is a `goto` statement; however, we do *not* want to optimise this by replacing it by a direct jump as this would cause problems for the decompilation procedure (see §5).

4.7 Method definitions

These are now easy to compile. We emit a JVM method definition whose bytecode consists of the code obtained by compiling the initial value declarations. This is followed by code to compute the result, and then by blocks of code corresponding to any local function definitions.

4.7.1 Metadata

To simplify the decompilation process we attach some metadata to the class file. Firstly, we record the names and types of the variables corresponding to the JVM registers. The JVM class file format provides a standard attribute for storing this information. This attribute allows one to specify the names and types of source-code variables associated with JVM local variables at various points within a method body. For instance, we could specify that a particular register represents an integer called `n` for the first 50 instructions and then represents a float called `currentApproximation` for the next 120 instructions. However, experimentation shows that this information is ignored by the JVM (or at least by Sun's standard implementation). If the metadata says that local variable 1 contains a float throughout a particular method then it is possible that it actually contains a value of type `java.lang.String`, and verification will still succeed. In view of this, we may need to carry out our own verification of the metadata.

We also use a user-defined attribute (`uk.ac.ed.dcs.mrg.GrailFunctionInfo`) which records the names of the local functions together with the numbers of the registers containing their arguments. This information is stored sequentially in the same order as the local functions occur. It is debatable whether we really require this information. Grail argument lists are essentially just annotations describing the registers which are "live" on function entry, and this could be determined by examining the bytecode. With the present scheme it is also con-

ceivable that someone could forge the function argument information, although this would be caught during post-decompilation typechecking. Again, some extra verification may be required.

5 Decompilation

Decompilation is essentially a process of parsing the bytecode in method bodies. This is really just a question of reversing the process described in the previous section, and we will not go into too much detail.

5.1 The decompilation process

We make an initial pass through the bytecode during which we label the destinations of all jumps. These are marked as being either *function labels* or *if-labels* depending on whether they are reached from a `goto` statement or some other statement such as `if_icmpeq` (this is why we resisted an optimisation in the previous section: we require that each label belong to precisely one of these categories).

After this we scan the bytecode looking for function definitions. We begin by looking for primitive operations corresponding to initial value declarations.

5.1.1 Primitive operations

We proceed by means of a sort of abstract evaluation. We scan through the bytecode consuming operations which load values onto the stack, and keep a record of the variables and constants which are on the stack, together with their types. The basic idea is that when we reach an instruction which does not correspond to data being loaded onto the stack we assume that we have reached a primitive operation. If the following instruction doesn't correspond to a Grail primitive operation then we assume that we're looking at code which computes a function result, and attempt to confirm this (see next section). Otherwise, we check that the correct number and type of values are on the stack. If the stack data does not match the primitive operation then the classfile is rejected as not containing valid Grail. If the stack data and the primitive operation do match then we use the stack information to construct an appropriate value declaration. If the primitive operation returns a value then this will involve checking that the next instruction is a store instruction assigning the result to a suitable variable. Having constructed the piece of Grail abstract syntax corresponding to the primitive operation we save it and begin again.

5.1.2 Function results

At this point have have encountered an instruction which does not correspond to a Grail primitive operation and we possibly have some data on the stack.

If the current instruction is a return instruction then we check that the correct amount of data is on the stack (rejecting the class file if not) and record

a corresponding primitive result. If the instruction is a `goto` instruction then we check that the stack is empty and that the destination of the jump is a function label (as determined in the initial pass); if these conditions are satisfied then we record a result which is a function call and go on to the next function.

The final case occurs when we have a comparison instruction. In this case we must check that we have two variables of the correct type on the stack. As above, we then look for a primitive result. If successful, we check that the comparison instruction branches to the point immediately after the first result (we don't want there to be any extra code in the space between the two results), and then look for the second primitive result. Assuming that no problems have arisen we proceed to the next step.

5.1.3 Function declarations

By this point we have a list of value declarations and a result. We now look up the name and argument list of the current function in the metadata and combine it with the value declarations and the result to form a complete function declaration. (Actually, we have to be slightly careful here: the first "function declaration" which we encounter requires special treatment since it is actually a method declaration, but the procedure is basically the same).

5.1.4 Method definitions

We now check whether there is any bytecode left; if so, we check that the current instruction is associated with a function label (*ie* that it really is the start of the bytecode for a function body) and begin the whole process again. When we finally run out of bytecode we assemble our collection of function declarations into a method definition.

5.1.5 Class definitions

Having processed all of the method definitions in the classfile we combine them with field definitions and various pieces of metadata specifying access permissions and the like to form an entire Grail abstract syntax tree. As a final check for consistency this can be fed to the Grail typechecker.

5.2 Compilation and decompilation

We claim that if a Grail program compiles (and verifies) successfully then decompiling the resulting class file produces a Grail program which is essentially identical to the original source program. There may be small differences since an expression such as

```
let
   val () = putstatic <int Class.field> 5
in () end
```

will compile to the same bytecode as

```
let
in putstatic <int Class.field> 5 end
```

but for Grail programs in an appropriate normal form the compilation/decompilation process will be the identity function. We hope to produce a theorem giving a precise description of the situation at a later date.

6 What about .NET?

Since the instruction set of the .NET CLR contains a subset (.Grail) which is very similar to the subset of the JVM instruction set used here it should be reasonably easy to compile Grail for .NET (see [10]). To accomplish this with the least amount of effort the simplest tactic might be to retain the Java class file format as the Grail transmission format and implement a separate translator which converts suitable Java class files into .NET assemblies prior to execution.

Appendix A: Implementations

A Implementations

The schemes for compilation and decompilation described above have been implemented and are available in the MRG repository in `progs/Grail`. The Grail-to-JVM compiler is called `gdf` (Grail defunctionaliser) and the JVM-to-Grail decompiler is called `gf` (Grail functionaliser). The choice of names is an attempt to suggest that the functional and bytecode versions of Grail are really equivalent, with neither being the "real" form.

A.1 The compiler

The `gdf` complier is written in Moscow ML (`www.dina.dk/~sestoft/mosml.html`), and requires that Moscow ML be installed on the user's system in order to work. `gdf` uses the sml-jvm toolkit of P. Bertelsen ([2]; see also `www.dina.dk/~pmb`) to construct bytecode and classfiles.

The `gdf` directory contains a precompiled version which should run on Linux systems. On other systems, change to the `src` directory and type "make"; this should produce an executable `gdf` program.

To use the compiler, simply write Grail source code into a text file and then type `gdf` *`name-of-file`*. For example, if a file `test.gr` contains a Grail definition of a class `T` then typing `gdf test.gr` should produce a JVM class file called `T.class` in the current directory.

`gdf` adds one feature to Grail for convenience. At the top of a source file (before any class definitions) one can write statements of the form

```
alias BigInt = java.math.BigInteger
```

Any occurrences of `java.math.BigInteger` can then be replaced by `BigInt`, allowing one to write for example

```
val b = invokevirtual b
     <BigInt BigInt.multiply(BigInt)> (n)
```

instead of

```
val b = invokevirtual b
     <java.lang.BigInteger
         java.lang.BigInteger.multiply(java.lang.BigInteger)> (n)
```

This can be used to produce substantial improvements in the readability of Grail code.

The error messages produced `gdf` are not as informative as they might be (for example, no line numbers are given for errors). This is because we assume that a large part of `gdf` will be used as the back-end for a Camelot-to-JVM compiler, and in this situation no textual Grail source code will ever exist. The present implementation of `gdf` is intended largely for experimentation, and we do not expect that large Grail programs will be compiled with it.

A.2 The decompiler

The `gf` complier is written in Standard ML using MLj (`www.dcs.ed.ac.uk/~mlj/`) and makes use of the Byte Code Engineering Library (BCEL - see `bcel.sourceforge.net` and [3]) of M. Dahm to disassemble JVM classfiles. MLj allows one to write Standard ML programs which are translated into JVM classes, and also to interface SML with arbitrary Java classes. BCEL consists of a large collection of Java classes which can be used to manipulate Java bytecode and classfiles. We have used the combination of BCEL and MLj for two main reasons:

- The sml-jvm toolkit can only be used to construct class files; no provision is made for deconstruction. Because of this we have chosen to deconstruct classfiles using BCEL.
- We imagine that some version of the decompiler will eventually be used as part of the proof-carrying code system, and will have to run on the machines of arbitrary consumers. Such consumers will presumably have a JVM installed on their machine (since they will be attempting to run Grail programs which have been complied into JVM bytecode), and so they will not require any extra software to run the decompiler. Unfortunately, such users will require to have the BCEL installed.

This situation is not ideal. The BCEL is large and complex, and a considerable amount of work is required to convert from the BCEL representation of a classfile into the sml-jvm format used in the Grail abstract syntax. This makes the decompiler much slower than the compiler, by a factor of approximately 14. It is hoped that at a later date we may extend the sml-jvm toolkit to allow

deconstruction of classfiles. This would allow the decompiler to be written in Moscow ML, which would lead to considerable increases in speed and remove the BCEL from our trusted computing base. If required, it would still be possible to convert the decompiler into JVM bytecode by using MLj. However, these matters are not a priority at the present moment.

To compile `gf` both MLj and the BCEL must be installed on the user's system; to run the precompiled version (in the top level of the `gf` directory, only BCEL is required. The precompiled version of `gdf` consists of a collection of JVM classes contained in the file `gf.zip`. To use this, the files `bcel.jar` and `gf.zip` must be in the user's classpath. The user must type `java gf` ⟨*classname*⟩, where ⟨*classname*⟩ is the name of the class to decompile (which must also be in the user's classpath). If the class is valid Grail then it will be decompiled into Grail source code which will be sent to standard output, whence it can be redirected to a file if desired. The file `gf` contains a small shell script which (on the Edinburgh Linux systems) automates the process of placing BCEL and `gf.zip` in the classpath and invoking the appropriate `java` command.

A.3 Examples

The `GrailExamples` subdirectory contains a few simple examples of Grail programs, along with some Java programs to test the Grail classes.

References

[1] L. Beringer. Core Grail. LFCS, University of Edinburgh, 2002. Available at `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

[2] P. Bertelsen. Compiling SML to Java bytecode, 1998.

[3] Markus Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[4] ECMA. Standard ECMA-335: Common language infrastructure (CLI). See `http://www.ecma.ch`.

[5] Christopher League. *A Type-Preserving Compiler Infrastructure*. PhD thesis, Yale University, December 2002. Available at `http://flint.cs.yale.edu/flint/publications/league-diss.html`.

[6] 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.

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

[8] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999. Available at `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.ht%ml`.

[9] K. MacKenzie. From Camelot to Grail: Compiling a high-level language. LFCS, University of Edinburgh, 2002. Available at `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

[10] K. MacKenzie. Grail and .NET. LFCS, University of Edinburgh, 2002. Available at `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

[11] K. MacKenzie. A virtual machine platform for resource-bounded computation, version 1.3. LFCS, University of Edinburgh, 2002. Available at `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[13] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] O. Shkaravska. High level functional language for resource-bounded computation (version 2.1.2). Ludwig-Maximilians-Universität München, 2002. Available at `http://www.dcs.ed.ac.uk/home/mrg/publications/`.

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