Deep type inference for mobile functions

Stephen Gilmore

Laboratory for Foundations of Computer Science
The University of Edinburgh
King's Buildings
Edinburgh EH9 3JZ
Scotland.

Telephone: +44 (0)131-650-5189.
Fax: +44 (0)131-667-7209.
Email: Stephen.Gilmore@ed.ac.uk

Abstract. We consider the problem of assessing the trustworthiness of mobile code. We introduce the idea of *deep type inference* on compiled object code and explain its usefulness as a method of deciding the level of security management which a unit of mobile code will require.

1. INTRODUCTION

The mobile agent paradigm is emerging as a leading programming paradigm for the next generation of networked computing architectures. A mobile agent can be deployed for evaluation of a computation on a remote host. This remote evaluation can both improve data locality and make economical use of available network resources. However, the only reasonable security policy for a computational resource provider to adopt is one which considers all computations of non-local origin to be potentially hostile. Thus programming languages such as Java[1] [1] where the notion of mobility is native to the language enforce *sandboxing* of mobile code. This prevents anti-social behaviour such as the creation and deletion of local files, sending and receiving electronic mail and making network connections other than back to the point of origin of the mobile code. The Java language provides degrees of programmer-definable control over the amount of liberty which mobile code is allowed. The programming abstraction in the Java language which is responsible for enforcing the sandboxing of non-native code is called a *security manager*.

In those cases where the code comes from a trusted host, it may be possible to allow the degree of sandboxing to be relaxed by installing a more liberal security manager. However, in the general case, it is still always necessary in Java to apply complete sandboxing to mobile code from an untrusted source. This sandboxing, and the degree of attendant indirection of execution of system functions which goes with it, only serves to be an unnecessary computational overhead in the cases where the mobile code actually has no potential for harmful behaviour.

The same reasoning applies in the case of the use of an untrusted library of potentially side-effecting functions. A mobile agent might wish to exploit data locality by using a local copy of a library. However, if the agent also wishes to retain control of its state it could employ a *state manager* which passes copies of mutable data values to library functions, discarding these copies on completion of the function invocation, on the assumption that they may have been altered by a side-effect of the function invocation. In the object-oriented programming model objects are routinely passed as parameters to method invocations. Here, the attendant object cloning and production of garbage which will require collection later could impose a significant performance penalty on a mobile agent. As was the case with the use of a Java security manager, this attendant performance penalty is entirely unnecessary when the local copy of the library actually has no potential for harmful behaviour.

The detection of harmful behaviour can be formulated as a *type inference* problem which is applied to the object code which is produced from some high-level source. We term this *deep* type inference in contrast to the *shallow* type inference which is performed on source code expressed in high-level languages such as Standard ML [2]. The *deep types* which are produced from this type inference are considerably more complex than traditional shallow types and are even more complex than those which are produced by type and effect inference. One

[1] Java is a trademark of Sun Microsystems.

reason why deep types may be allowed to be so complex is that they are purely internal and serve as an abstract typing valuation within the run-time interpreter. Thus, in pointed contrast to the shallow types of Standard ML, a deep type is never seen by an application programmer.

We present examples in the setting of Java byte code which show that this form of type inference is applicable to byte code which is either of functional or imperative origin. We use the MLj compiler [3] to compile Standard ML code to Java byte code and compare this with the code which is produced from Java source by Sun's `javac` compiler. We show that even untrusted code which has functional *essence*, rather than just functional syntax, can be allowed to run unrestricted without incurring the overhead of a security manager. The benefits which arise from the functional programming paradigm are seen to come from the disciplined control of state. The use of a purely functional language can be seen as one which has entirely suppressed the use of state.

2. COMPILING TO JAVA BYTE CODE

There are a number of competing strategies for executing Java byte code. The primary method of execution is to interpret the byte code although another is *just-in-time* compilation, which compiles it to native machine code for native execution. Yet another is a combination of interpretation and compiled code execution as directed by feedback from continuous performance profiling. This technique is used in Sun's Java HotSpot system [4]. Regardless of their execution policy, platforms which execute Java byte code are known as Java Virtual Machines (JVMs) if they conform to the JVM specification [5].

The widespread availability of implementations of the Java Virtual Machine has encouraged implementors of other programming languages to target Java byte code as a form of portable assembly language. Of interest to the functional programming community in particular are compilers which produce Java byte codes from Standard ML [3, 6] and from Haskell [7, 8].

Comparing compilation units[2] for Standard ML and Java, as shown in Figure 1, we have a Standard ML *structure* in one case and a Java *class* in the other. A Standard ML structure may be accompanied with a *signature* which identifies the components of the structure which are to be accessible outside the structure body via long identifiers such as the function `Fac.fact` in this case. The Java programming language provides control of visibility and encapsulation through the use of the *access control modifiers* `public`, `private` and `protected`.

[2] The term *compilation unit* is a little misleading in the context of MLj because it does not provide a full separate compilation facility. An extension to the system is planned which would provide some persistent storage of intermediate compiled forms but at the time of writing, the MLj compiler operates as a *whole-program* compiler. As such it is able to apply more aggressive type-directed optimisation than a separate compiler would. The subject of cross-module optimisation is the subject of recent research in the Standard ML community [9].

```
signature Fac =
sig
  val fact : int -> int
end;

structure Fac :> Fac =
struct
  fun fac (n, m) =
    if n = 0
    then m
    else
      fac (n - 1, m * n)

  fun fact n =
    fac (n, 1)
end;
```

```
class Fac {
  private int m;

  private int fac (int n, int m) {
    while (n != 0) {
      m *= n;
      n--;
    }
    return m;
  }

  public int fact (int n) {
    m = 1;
    return fac (n, m);
  }
}
```

Fig. 1. A Standard ML structure and a Java class

The Java class defines a method `fact()` which object instances of the class `Fac` will provide.

Of the two code extracts which are included in Figure 1, neither are idiomatic in the sense that they are the implementations would have been produced by programmers who routinely work in Standard ML or Java. In the Standard ML case it would be usual for an implementor to define the `fac` function by pattern matching, exploiting this elegant feature of the language to separate out the two cases in the function definition. In the Java example, the local variable `m` does not serve any very useful purpose and decomposing the method `fac()` does not provide any useful structure since this method is implemented iteratively, not using a tail recursive function as in the Standard ML case. However, the two example code fragments are both *representative* in the sense that the Standard ML example contains no assignments and defines the function recursively whereas the Java version uses updates and a loop to compute the same results (ignoring different behaviour on numeric overflow).

The two code fragments also have a common point of comparison in the function `fac` and the method `fac()`. Neither are visible outside their respective compilation units although they do of course occupy space in their compiled representations. A Java disassembler such as Sun's `javap` provides a convenient way to inspect these compiled representations. A representative extract of the byte code produced from these compilation units is shown in Figure 2.

As might be expected, it becomes difficult after compilation to determine which bytecodes resulted from the functional Standard ML input and which resulted from the imperative Java input. Both compiled representations include loads (`iload` instructions) and stores (`istore` instructions) and both contain

```
Method int fac(int,int)
   0 goto 19          14 istore_1
   3 iload_1          15 iload_0
   4 ireturn          16 iconst_m1
   5 iconst_0         17 iadd
   6 istore_2         18 istore_0
   7 iload_2          19 iload_0
   8 ifne 3           20 ifne 5
  11 iload_1          23 iconst_1
  12 iload_0          24 istore_2
  13 imul             25 goto 7
```

```
Method int fac(int,int)
   0 goto 10
   3 iload_2
   4 iload_1
   5 imul
   6 istore_2
   7 iinc 1 -1
  10 iload_1
  11 ifne 3
  14 iload_2
  15 ireturn
```

Fig. 2. Java byte code extracts

conditional and unconditional jumps (the `ifne` and `goto` instructions). In fact, the bytecodes on the left come from the Standard ML source and the bytecodes on the right come from the Java source[3].

In order to see how deep type inference can allow us to recover the functional essence from byte code which seems as though it might have been produced from imperative source code we first need to understand the role of types in the Java Virtual Machine. We discuss this subject in the following section.

3. TYPES AND THE JVM

The Java Virtual Machine is a typed abstract machine. The types which it directly supports correspond to a subset of the Java types. Separate instructions implement conversions between the unsupported types and the supported types. The design decision which forces this distinction to be drawn is the wish to represent every JVM opcode by a single byte. If typed instructions were to be provided for each Java type there would be too many to be represented in a byte. Thus, although the methods `not()` and `neg()` in Figure 3 below have different parameter and return types the JDK 1.1.6 Java compiler emits identical bytecodes for their method bodies[4]. Note that additional type information is stored independently of the bytecodes. This is the information which is retrieved via Java's *reflection* capabilities.

It is not only booleans which the JVM represents as integers. Characters are also zero-extended to 32-bit integers and 8 and 16-bit integers are sign-extended

[3] It might have been possible to guess this due to the use of the `iinc` instruction with arguments (local variable number) 1 and increment -1. This corresponds directly to the Java command `n--` which has no analogue in Standard ML. The parameter passing convention in the JVM is that the first parameter to a method is stored into the first local variable for a non-static method and in the zeroth local variable for a static method.

[4] The Java compiler in Sun's Java 2 SDK will generate more compact byte code for the `not()` method by deploying an `ixor` instruction on the parameter `b` and the constant 1 but it still encodes boolean values as integers.

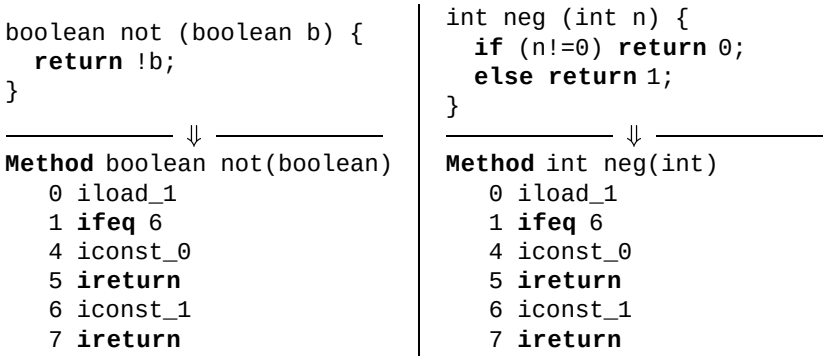

Fig. 3. Boolean values are manipulated as integers in the JVM

to 32 bits. The correspondence between storage types in the languages Standard ML and Java and computation types in the JVM is shown in Figure 4.

Standard ML type	Java type	JVM type
`bool`	`boolean`	`int`
`char`	`char`	`int`
`Int8.int`	`byte`	`int`
`Int16.int`	`short`	`int`
`int`	`int`	`int`
`Int64.int`	`long`	`long`
`Real32.real`	`float`	`float`
`real`	`double`	`double`

Fig. 4. Storage types and computational types in the JVM

4. DEEP TYPES AND THE JVM

The conclusion is that we evidently cannot reconstruct the Java type of a method from its compiled bytecodes. Neither is it possible to reconstruct the principal type of a Standard ML function since the JVM provides no support for the expression of parametric polymorphism in routines. However, we are seeking to establish here instead the presence or absence of potential side-effects in the execution of a compiled JVM bytecode sequence and to identify the object fields which could potentially be modified by a method call. We term the formal expression of this information the *deep type* of a method.

We identify a representative subset of the Java bytecodes which we term JVML$_d$. The operational semantics of this subset is presented in Figure 5. Tuples

of machine states contain a program counter pc, a total map f which maps local variables from the set VAR to values, and an operand stack s.

$$\frac{P[pc] = \texttt{inc}}{P \vdash \langle pc, f, n \cdot s\rangle \to \langle pc+1, f, (n+1) \cdot s\rangle}$$

$$\frac{P[pc] = \texttt{pop}}{P \vdash \langle pc, f, v \cdot s\rangle \to \langle pc+1, f, s\rangle} \qquad \frac{P[pc] = \texttt{push}\ 0}{P \vdash \langle pc, f, s\rangle \to \langle pc+1, f, 0 \cdot s\rangle}$$

$$\frac{P[pc] = \texttt{load}\ x}{P \vdash \langle pc, f, s\rangle \to \langle pc+1, f, f[x] \cdot s\rangle} \qquad \frac{P[pc] = \texttt{store}\ x}{P \vdash \langle pc, f, v \cdot s\rangle \to \langle pc+1, f[x \mapsto v], s\rangle}$$

$$\frac{P[pc] = o.\texttt{getfield}\ x}{P \vdash \langle pc, f, s\rangle \to \langle pc+1, f, f[o][x] \cdot s\rangle}$$

$$\frac{P[pc] = o.\texttt{putfield}\ x}{P \vdash \langle pc, f, v \cdot s\rangle \to \langle pc+1, f[o \mapsto o[x \mapsto v]], s\rangle}$$

$$\frac{P[pc] = \texttt{if}\ L}{P \vdash \langle pc, f, 0 \cdot s\rangle \to \langle pc+1, f, s\rangle} \qquad \frac{P[pc] = \texttt{if}\ L \qquad n \neq 0}{P \vdash \langle pc, f, n \cdot s\rangle \to \langle L, f, s\rangle}$$

$$\frac{P[pc] = \texttt{goto}\ L}{P \vdash \langle pc, f, s\rangle \to \langle L, f, s\rangle} \qquad \frac{P[pc] = \texttt{new}\ \sigma \quad o_l \in \mathcal{O}^{\sigma_{pc}} \quad \mathit{Unused}(o_l, f, s)}{P \vdash \langle pc, f, s\rangle \to \langle pc+1, f, o_l \cdot s\rangle}$$

Fig. 5. JVML$_d$ operational semantics

We use the variables σ to range over any object type and write $\mathcal{O}^\sigma$ for the set of values of that type. A particular object value from that set will be denoted by o or o_l. We use o_l solely to denote locally generated object accessors, meaning that they are generated by this method invocation instead of being passed in as parameters[5]. We write $\mathit{Unused}(o, f, s)$ to abbreviate $o \notin s \wedge o \notin \mathit{Rng}(f)$.

In defining the static semantics of the language we begin by identifying semantic objects which shadow the roles of the state function f and the operand stack s which are found in the dynamic semantics. We name the corresponding static semantic objects F and S. The former is a mapping from addresses to functions which map local variables to types. The latter is a mapping from addresses to stack types. Thus, $F_i[y]$ is the type of local variable y at line i of the program and S_i is the type of the operand stack at the same place.

[5] Java calls by value. In passing objects to methods it is important to distinguish between passing an object reference by value and passing an object value by reference. Java does the former and not the latter.

$$\frac{P[i] = \texttt{inc} \quad F_{i+1} = F_i \quad S_{i+1} = S_i = \textsc{Int} \cdot \alpha \quad D_{i+1} = D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{if}\ L \quad F_{i+1} = F_L = F_i \quad S_i = \textsc{Int} \cdot S_{i+1} = \textsc{Int} \cdot S_L \quad D_{i+1} = D_L = D_i \quad i+1 \in \mathit{Dom}(P) \quad L \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{pop} \quad F_{i+1} = F_i \quad S_i = \tau \cdot S_{i+1} \quad D_{i+1} = D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{push}\ 0 \quad F_{i+1} = F_i \quad S_{i+1} = \textsc{Int} \cdot S_i \quad D_{i+1} = D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{load}\ x \quad x \in \mathit{Dom}(F_i) \quad F_{i+1} = F_i \quad S_{i+1} = F_i[x] \cdot S_i \quad D_{i+1} = D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{store}\ x \quad x \in \mathit{Dom}(F_i) \quad F_{i+1} = F_i[x \mapsto \tau] \quad S_i = \tau \cdot S_{i+1} \quad D_{i+1} = D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = o.\texttt{getfield}\ x \quad o \in \mathit{Dom}(F_i) \quad x \in \mathit{Dom}(F_i[o]) \quad F_{i+1} = F_i \quad S_{i+1} = F_i[o][x] \cdot S_i \quad D_{i+1} = \mathsf{rd}(o, x, F_i[o][x]) \cdot D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = o.\texttt{putfield}\ x \quad o \in \mathit{Dom}(F_i) \quad x \in \mathit{Dom}(F_i[o]) \quad F_{i+1} = F_i[o \mapsto o[x \mapsto F_i[o][x]]] \quad S_i = F_i[o][x] \cdot S_{i+1} \quad D_{i+1} = \mathsf{wr}(o, x, F_i[o][x]) \cdot D_i \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{halt}}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{goto}\ L \quad F_L = F_i \quad S_L = S_i \quad D_L = D_i \quad L \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

$$\frac{P[i] = \texttt{new}\ \sigma \quad F_{i+1} = F_i \quad S_{i+1} = \sigma_i \cdot S_i \quad D_{i+1} = D_i \quad \sigma_i \notin S_i \quad \sigma_i \notin \mathit{Rng}(F_i) \quad i+1 \in \mathit{Dom}(P)}{F, S, D, i \vdash P}$$

Fig. 6. JVML$_d$ static semantics

To this we add a static semantic object to accumulate the deep typing information from the method body. This records the field accesses and stores of objects which are accessed by this method invocation. We regulate the correct use of the program counter by checking the successful progression through to the next instruction in most cases. Exceptionally, the `goto` and `halt` instructions do not need this test since the immediately following instruction is not reached in either case. The `if` instruction has two possible destinations, both of which must be checked. This gives rise also to two possible subsequent semantic values for each of the functions for state typing, stack typing and deep typing.

5. RELATED WORK

Because security properties must ultimately be verified on Java bytecode the importance of thoroughness here was identified early by authors interested in the subject [10]. The absence of a formal description of the type system for Java bytecode was an obvious source of concern. Another was the deviation of Java bytecode type-checking from traditional type-checking where the type-correctness of a construct depends upon the current typing context, the type-correctness of subexpressions, and whether the construct is typable by one of a fixed set of rules. In contrast, the Java bytecode verifier must show that all possible execution paths lead to the same virtual machine configuration.

A compelling type system for Java bytecode subroutines has previously been given by Stata and Abadi [11]. Their main theorem shows that for methods expressed in a subset of Java's bytecodes (JVML0) when method execution stops it is because of a `halt` instruction and not program counter overflow or violation of an instruction precondition. Further, the operand stack, with the return value on top, is well-typed. This work has been continued by Hagiya and Tozawa [12] and by Freund and Mitchell [13] leading in the latter case to the detection of a previously unknown bug in the Sun JDK 1.1.4 bytecode verifier. Other work by Qian [14] is being developed and may lead to the first provably-correct implementation of the JVM bytecode verifier [15].

Ours is the first attempt to consider *interference* properties [16] in the types which are inferred for Java bytecode subroutines. The application of type inference to a low-level, imperative language such as Java byte code might seem to be ill conceived. However, although type inference has primarily found favour in the functional language community, there is no reason why it cannot be applied to imperative languages. Successful examples of such application include dialects of Pascal [17] and C [18].

Acknowledgements. Stephen Gilmore is supported by the 'Distributed Commit Protocols' grant from the EPSRC and by Esprit Working group FIREworks.

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Second edition, 1998.
2. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.
3. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Third ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, Baltimore, 1998.
4. D. Griswold. The Java Hotspot Virtual Machine Architecture. SUN Microsystems White Paper, March 1998.
5. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, 1999.
6. N. Benton and A. Kennedy. Interlanguage working without tears: Blending SML with Java. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999.
7. J. Peterson and K. Hammond, editors. Haskell 1.4: A non-strict purely functional language. The Haskell Committee, April 1997.
8. D. Wakeling. Mobile Haskell: Compiling lazy functional programs for the Java Virtual Machine. In *Proceedings of the 1998 Conference on Programming Languages, Implementations, Logics and Programs (PLILP'98)*, volume 1490 of *LNCS*, pages 335–352. Springer Verlag, September 1998.
9. M. Blume and A. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 112–124, June 1997.
10. D. Dean, E.W. Felten, and D.S. Wallach. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.
11. R. Stata and M. Abadi. A type system for Java bytecode subroutines. Technical Report 158, Digital Equipment Corporation Systems Research Center, June 1998. To appear in ACM Transactions on Programming Languages and Systems.
12. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. In *SIG-Notes, PRO-17-3*, pages 13–18. Information Processing Society of Japan, 1998.
13. S. Freund and J.C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Symp. Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*, 1998.
14. Z. Qian. *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines*, chapter 8 of *Formal Syntax and Semantics of Java*. Springer-Verlag LNCS 1523, 1999.
15. A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. Kestrel Institute, Palo Alto, California, July 1998.
16. R.D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983.
17. O.I. Hougaard, M. Schwartzbach, and H. Askari. Type inference of Turbo Pascal. *Software—Concepts and Tools*, (16):160–169, 1995.
18. G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.