Chapter 1

Deep Type Inference for Mobile Functions

Stephen Gilmore[1]

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.

1.1 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 [AG98] 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

---

[1] Laboratory for Foundations of Computer Science, The University of Edinburgh, King’s Buildings, Edinburgh, EH9 3JZ, Scotland; Phone: +44 (0)131-650-5189; Fax: +44 (0)131-667-7209; Email: `Stephen.Gilmore@ed.ac.uk`

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 inference of *deep* types in contrast to the inference of *shallow* types performed on source code expressed in high-level languages such as Standard ML [MTHM97]. The *deep types* which are produced from this type inference extend traditional shallow types by including a static representation of any reads and stores which may be performed on non-local variables. For any pure function, the shallow and deep types will be equal but for an impure function they will not. Deep types 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 [BKR98] 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 assigns only to local variables 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.

1.2 COMPILING TO JAVA BYTE CODE

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 [BKR98, BK99] and from Haskell [PH97, Wak98]. Comparing compilation units for Standard ML and Java, as shown in Figure 1.1,

```
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);
  }
}
```

FIGURE 1.1. A Standard ML structure and a Java class

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`. The Java class defines a method `fact()` which object instances of the class `Fac` will provide. The two example code fragments in Figure 1.1 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 1.2.

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)              | Method int fac(int,int)
   0 goto 19       14 istore_1       |    0 goto 10
   3 iload_1       15 iload_0        |    3 iload_2
   4 ireturn       16 iconst_m1      |    4 iload_1
   5 iconst_0      17 iadd           |    5 imul
   6 istore_2      18 istore_0       |    6 istore_2
   7 iload_2       19 iload_0        |    7 iinc 1 -1
   8 ifne 3        20 ifne 5         |   10 iload_1
  11 iload_1       23 iconst_1       |   11 ifne 3
  12 iload_0       24 istore_2       |   14 iload_2
  13 imul          25 goto 7         |   15 ireturn
```

FIGURE 1.2. 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.

In order to see how deep type inference can allow us to distinguish code with the potential to update non-local variables from code with no such potential we first need to understand the role of types in the Java Virtual Machine. The Java Virtual Machine is a typed abstract machine. The types which it directly supports correspond to a subset of the Java types. We 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 will term JVML$_d$. The operational semantics of this subset is presented in Figure 1.3. Tuples of machine states contain a program counter i, a total map f which maps local variables from the set VAR to values, and an operand stack s.

We use the variables σ to range over any object type and write O^σ 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. We write $Unused(o, f, s)$ to abbreviate $o \notin s \wedge o \notin Rng(f)$.

In defining the static semantics of the language as presented in Figure 1.4 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

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

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

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

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

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

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

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

FIGURE 1.3. JVML$_d$ operational semantics

at line i of the program and S_i is the type of the operand stack at the same place.

Our interest here has been in formulating the deep type for a function or method. We use the variable D to range over deep typings. These are purely static semantic objects which have no counterpart in the dynamic semantics. When we use such a static semantic object to accumulate the deep typing information from the method body we manipulate it as a stack onto which information is only pushed. The stack never decreases in size as we investigate the instructions in the body of the method. It is possible that no information is added as we go from line i to the following line ($D_{i+1} = D_i$). In the significant cases we have $D_{i+1} = d \cdot D_i$ where pushing d onto the top of the stack records either reading field x of object o at a particular type or writing field x of object o at a particular type. The tags rd and wr are used to distinguish reading from writing.

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.

1.3 UNDERSTANDING THE STATIC SEMANTICS

To explore the static semantics further we select one of the rules to consider in greater depth. As our illustrative example we consider the rule for the `putfield` instruction.

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

This rule is applicable when instruction i of the program P (denoted $P[i]$) is $o.\texttt{putfield}\ x$. This instruction pops the value on top of the operand stack S and writes it to field x of object o. Of course, o must be an object in the store typing at line i (that is, $o \in \mathit{Dom}(F_i)$) and x must be one of its fields at this position (that is, $x \in \mathit{Dom}(F_i[o])$), treating objects as environments themselves.

The value on top of the operand stack will have some type, say τ, and the operand typing stack will be popped by this instruction (that is, $S_i = \tau \cdot S_{i+1}$). Because the value of this type is written to field x of object o this will cause a change in the store typing for the next instruction so that the typing associated with x in o will now be τ. The new typing for o itself is $o[x \mapsto \tau]$ and the entry for o in F_i is updated to reflect this change.

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

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

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

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

$$\frac{P[i] = \texttt{halt}}{F,S,D,i \vdash P} \qquad \frac{\begin{array}{c} P[i] = \texttt{goto}\ L \\ F_L = F_i \\ S_L = S_i \\ D_L = D_i \\ L \in \mathit{Dom}(P) \end{array}}{F,S,D,i \vdash P} \qquad \frac{\begin{array}{c} P[i] = \texttt{new}\ \sigma \\ F_{i+1} = F_i \\ S_{i+1} = \sigma_i \cdot S_i \\ D_{i+1} = D_i \\ \sigma_i \notin S_i \\ \sigma_i \notin \mathit{Rng}(F_i) \\ i+1 \in \mathit{Dom}(P) \end{array}}{F,S,D,i \vdash P}$$

FIGURE 1.4. **JVML$_d$ static semantics**

We wish to record in the deep typing that a value of type τ can be written to field x of object o so we have a deep typing entry $\mathsf{wr}(o,x,\tau)$ for this. The stack of these typings is then increased to record this new information

$$D_{i+1} = \mathsf{wr}(o,x,\tau) \cdot D_i.$$

Finally, control will progress to the next instruction in the program, so it is important to check that this progression does not overflow the program counter (that is, $i+1 \in Dom(P)$).

1.4 USING DEEP TYPES TO DETECT UNCHECKED UPDATES

Having presented the static semantics for JVML$_d$ we are now able to consider the use of deep type inference on a simple example. In Figure 1.5 we present three different implementations of a simple function to add one to an integer reference value.

```
fun add1a x =
  !x + 1;
```

```
fun add1b x =
  (x := !x + 1;
   !x);
```

```
fun add1c x =
  let val y = ref(!x)
  in y := !y + 1; !y
  end;
```

FIGURE 1.5. Three Standard ML functions

The first, `add1a`, is a pure function. It dereferences an integer reference value (the Standard ML dereference operator is `!`) and returns the successor of the value which is stored in the reference cell. The second, `add1b`, has the side-effect of incrementing the reference value which it is passed so that successive calls of the function with the same argument will return different results. The third, `add1c`, makes a local copy of the reference value, increments this local copy, and returns the result.

Standard ML assigns to all three of these functions the same shallow type, namely `int ref -> int`. The supplementary deep typing information which we can derive from the compiled bytecode representation of these functions makes clear that the function `add1b` could not be used as a direct replacement for either of the others. The deep type for that function will include a record that field x of the first object parameter to the compiled representation of the function will be updated at type INT. By using deep type inference to detect potential updates of local state by code which has been downloaded from an untrusted source an application program can protect its correct functioning from potentially malicious updates to accessible store. In contrast, if deep type inference finds a unit of downloaded code to consist only of pure functions or impure functions which only update their own local state then such code may be allowed to execute without any supervisory overhead.

The deep type information which we derive is comprehensive in that it gives details about reads and updates of each argument to a function invocation. Even in call-by-value languages such as Standard ML and Java a function might use derefrencing to update one of its arguments. Since the deep type of the function will tell us exactly which arguments are (potentially) updated we can arrange to copy only those values which might be overwritten and restore their original values after the function call terminates. We can impose this form of protection from accidental or malicious update of our private storage locations economically because we copy only those values which might be overwritten. In the best case we can determine that no values will be overwritten and we need copy no values at all.

1.5 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 [DFW96]. 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 [SA98]. 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 [HT98] and by Freund and Mitchell [FM98] 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 [Qia99] is being developed and may lead to the first provably-correct implementation of the JVM bytecode verifier [CGQ98].

ACKNOWLEDGEMENTS

Stephen Gilmore is supported by the 'Distributed Commit Protocols' grant from the EPSRC and by Esprit Working group FIREworks. It is a pleasure to thank the anonymous referees for many insightful comments which led to improvements in this paper.

REFERENCES

[AG98] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Second edition, 1998.

[BK99] 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.

[BKR98] 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.

[CGQ98] A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. Kestrel Institute, Palo Alto, California, July 1998.

[DFW96] 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.

[FM98] 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.

[HT98] 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.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

[PH97] J. Peterson and K. Hammond, editors. Haskell 1.4: A non-strict purely functional language. The Haskell Committee, April 1997.

[Qia99] 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.

[SA98] 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.

[Wak98] 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.