Secure Execution of Mobile Java using Static Analysis and Proof Carrying Code

Robert Atkey (`bob.atkey@ed.ac.uk`)
Kenneth MacKenzie (`kwxm@inf.ed.ac.uk`)
School of Informatics, University of Edinburgh, UK

Christopher Paton (`chris@chrispaton.co.uk`)

Abstract

We consider the problems raised by the use of mobile code in e-Science. If a user submits a program for execution on a remote machine then problems can arise if the program misbehaves, for example by using too much memory or taking too long to execute. We describe methods which can be used to obtain unforgeable *a priori* guarantees that a mobile program will behave in a reasonable manner. We have a prototype implementation that executes mobile code securely within an OGSA-DAI server.

1 Introduction

A crucial aspect of e-Science is the transmission of executable code to be run on remote servers. Code migrates to environments that provide suitable resources for it to execute. Processor intensive code is sent to compute clusters or supercomputers that provide large amounts of processing power; code that works on massive data-sets is moved to the data rather than the other way round, it being far more efficient that way than to download data to the code. Often, code moves across administrative borders; a researcher from Institute A may want submit a job to be run against data stored by the University of B, where the two organisations have no prior relationship. The free movement of code across administrative borders is essential to the success of the Grid concept, but large security issues arise when one considers the running of untrusted code on one's expensive servers.

If remote users are allowed to execute such *mobile code* in a completely unrestricted way then all kinds of problems can arise for the server. Some of these are standard security problems (for example, remote users must be prevented from deleting files or using a machine for sending spam email) which can be dealt with by existing operating system security techniques. However, some issues are more difficult to deal with. For example, remote users may be allowed to create files on a server, but if too much disk space is used then other users may be unable to run their programs. Similarly, if a program uses too much memory, or runs for too long, then this may also inconvenience other users. Overuse of such resources[1] may be deliberate, as happens in a *denial-of-service attack*, but they may also be accidental (due to bugs in a program), or simply unanticipated (because the programmer can't tell in advance how their program will behave). Overuse of some resource by a mobile program may cause problems not just for other users, but also for the author of the program: if they submit a program which monopolises some resource then time or money may be lost with no benefit, and the programmer might even lose their rights to execute programs on a server if they cause too much inconvenience to others.

Current internet-accessible services that accept mobile code do of course attempt to protect themselves from malicious or otherwise unsafe code. The first line of defence in current practice is to authenticate the remote client in some way, ensuring that the code originates from a trusted source. The vetting of clients is intended to reduce the possibility of malicious code; and, should anything go wrong, the authentication

---

[1]The term *resource* is usually used in e-Science to refer to individual computers within a network or grid, but we use the word to refer to quantities which are consumed by a program during execution and which may be regarded as "costs" incurred by the program.

logs can be used to point the finger of blame.

For some applications, the burden of distributing authentication credentials to every potential user is too great, or the cost of failure is very high, so further measures must be taken to protect against badly behaved code. Typically, these take the form of *dynamic checks* on the executing code. This enforces good behaviour by monitoring the code as it runs; if the code does something that is not permitted then it is terminated and control returned to the server software. There are various techniques for accomplishing this, including running the code inside a virtual machine's sandbox, running it inside an entire virtualised machine, or by instrumenting the code with checks at critical points. The disadvantage of this approach is that problems are only dealt with at the time they would occur, and handling a violation of security policy typically results in termination or suspension of the process. This is not an ideal scenario several days into a complex batch job.

An alternative approach is to analyse the code before it is run. The techniques of *static analysis* [12] can be applied to programs (either source code or compiled object code) in order to predict some aspect of their behaviour *prior to execution*. The analysis can either be built in to a compiler or carried out by a standalone tool. Some analyses are fully automatic while others may require some participation by the programmer (for example, by adding annotations to the source code)

Static analysis could thus be used by a server to determine (conservatively) whether it would be safe to execute a mobile program which it had received: code which failed the test would not be run. A disadvantage of this approach is that static analyses are often computationally expensive, meaning that a server could spend a significant amount of time analysing code, rather than actually executing the code. Furthermore, static analyses can be complex and difficult to implement, meaning that either the analysis could be wrong or bugs in the analysis code could be exploited by a malicious attacker.

To mitigate these concerns, we use the technique of *proof carrying code* (PCC) [11]. With PCC, static analysis of the code is carried out by the client. This analysis produces evidence, in the form of a formalised, machine-checkable mathematical proof that is checked by the server before executing the code. Checking proofs is cheaper than producing them, and proof checkers are smaller and less complex than proof gen-

erators.

This paper is a description of the ongoing work of the ReQueST (Resource Quantification for e-Science Technologies)[2] project at the University of Edinburgh. This project aims to produce static analyses for Java programs that determine their resource usage and produce independently checkable proofs of the resource usage properties suitable for PCC.

1.1 Overview

Firstly we will describe a motivating scenario concerning Java code being sent to a large database to perform a custom query. We will base the rest of our discussion on this example. We will then discuss various methods which can be used to protect servers against the security risks inherent in the use of mobile code. After this, we describe a prototype implementation of these methods with the OGSA-DAI database access software. Finally we will discuss our plans for future work.

2 Motivating Scenario

In this paper we will concentrate on using Java bytecode as the format for the code that is to be sent by the client to be executed on the server. We have several reasons for this. Java bytecode is relatively portable between machine architectures and operating systems, meaning that a client does not have recreate the exact operating environment of the server in order to develop and compile their code. The use of a platform-independent compiled format rather than a source code format means that the server does not have to be complicated with the provision of compilers and the attendant tools in order to execute client's code. Also, the security-conscious design of Java means that it enjoys properties such as *type safety* and *memory safety* [6] which make it much more amenable to static analysis techniques than unsafe languages such as C and FORTRAN.

To motivate the use of mobile Java, we consider the following scenario. A large scientific database (of the order of hundreds of gigabytes) resides on a remote server, accessible via the internet. An e-Scientist wishes to query this data, extracting items of interest. Unfortunately, the criteria he wishes to use to filter the data – some

---

[2] `http://groups.inf.ed.ac.uk/request/`

image processing technique perhaps – is not expressible in the query language of the database management system that stores the data, but he does have it coded as a short Java program.

If he cannot upload the code to the server in order to process the data, our e-Scientist must download *all* of the data from the database to filter it, discarding the data that is not relevant. However, this may be impractical if the database is extremely large. The alternative method that we propose here is to modify the database server to allow the upload of the compiled Java code to the server where it can be run against the data. Only the filtered data (which will hopefully be much smaller than the full database) is sent back across the network.

3 Ensuring good behaviour

We consider several approaches to the problem of guaranteeing that a mobile program behaves well.

3.1 Runtime Monitoring

The simplest way to ensure good behaviour of a program is via runtime monitoring (or *dynamic monitoring*). Here, the program is executed in the normal way and its behaviour is monitored at it runs; if the program misbehaves by (for example) using too much memory or taking too long then it is terminated. This approach has several points in its favour:

- Most operating systems already have some built in means of process monitoring and control, so dynamic monitoring can be carried out with little extra overhead

- Programs are written in the usual way and no extra work is required on the part of the programmer

Dynamic monitoring is not without disadvantages however. In particular,

- By the time that it is realised that a program is misbehaving, it may already have used up valuable time or space, possibly causing a degradation of service for other users.

3.2 Static Analysis

Recall that static analysis involves analysing code before it is run. For properties such as memory usage, the static analysis approach

has several advantages over dynamic monitoring and profiling. Resource bounds are deduced from the program itself, and will be guaranteed to hold for all possible executions of the program (assuming that the analysis has been correctly designed and implemented). This enables a programmer to be confident that their program will be well-behaved before they submit it for remote execution, and they will therefore not be causing problems of the kind that may arise if dynamic monitoring is used. Also, the information obtained from static analysis can be helpful both to the programmer (perhaps enabling them to improve the behaviour of a program) and to the operators of the remote system (perhaps enabling them to make sensible scheduling decisions based on the expected behaviour of the program).

Many static analysis tools are available for Java (FindBugs[3], Klocwork[4] and ESC/Java2[5] for example) but most of them deal with program correctness (for instance, showing that a program never attempts to dereference a null pointer or access an array outside its bounds) rather than resource-related properties.

However, a considerable amount of research related to static prediction of resource bounds is currently ongoing, and prototype systems are beginning to appear. For example:

- The TINMAN system [10] uses a combination of static and dynamic techniques to predict and enforce bounds on execution time and memory usage for mobile C programs.

- In [7], Hofmann and Jost described a technique for automatically inferring memory usage bounds of functional programs. This was subsequently implemented in the Mobile Resource Guarantees project at the University of Edinburgh and LMU Munich [15, 8].

- The first author of the present paper has recently added memory specifications to the ESC/Java static analysis tool [3].

- The Mobius project[6] is presently developing methods for automatic analysis and verification of resource bounds for mobile

---

[3]http://findbugs.sourceforge.net/
[4]http://www.klocwork.com/
[5]http://secure.ucd.ie/products/opensource/ESCJava2/
[6]http://mobius.inria.fr/

code in the context of the Java MIDP platform[7] for mobile telephones. Static analysis techniques for resource usage have been studied at INRIA, Rennes [4, 5] and UPM, Madrid [1, 9].

3.3 Proof Carrying Code

The techniques of static analysis mentioned above are an improvement on dynamic techniques, but are still not entirely foolproof. For example, the following problems may arise:

- There may be errors in the design of an analysis technique.

- There may be errors in the implementation of an analysis tool.

- A remote user may claim that their program satisfies certain bounds, but there is no guarantee that these claims are correct. (This can be overcome by running the analysis on the server prior to execution, but this may involve an overhead).

These problems may be dealt with using the technique of *Proof-Carrying Code* (PCC) [11]. This involves constructing a *certificate* in the form of a formal, mechanically verifiable mathematical proof that a program P satisfies some property X. The proof is provided by the code producer (ideally automatically, by an enhanced static analyser) and packed together with the program before being sent to the consumer. The consumer then checks the program against the proof, and – if the check is successful – can be assured that the program does indeed satisfy the given property. This provides an *unforgeable* guarantee: either the proof is valid and does indeed demonstrate that P satisfies the property X, or it is invalid, and the code consumer can reject P and refuse to run it. Problems of incorrectness in the design or implementation of the analysis are also overcome: if for some reason the analysis comes up with an incorrect answer then it will be impossible to construct a valid proof that the answer is actually correct. Figure 1 contains a schematic illustration of a PCC system.

The advantages of PCC are obtained at a cost, however. Implementing a PCC system involves designing a mechanisable mathematical formalisation of the properties of the language in which one is interested (a *formal semantics* [17]), and this requires considerable time and

expertise. Some means has to be found of expressing the properties in which one is interested, and this may require the creation of a specialised *program logic* which then has to be shown to be correct with respect to the semantics. It is also necessary to find a means of producing proofs automatically during the process of program analysis. However, these steps have to be carried out only once, and the PCC system can then operate automatically, with little or no intervention required from the programmer.

The PCC paradigm can provide very strong guarantees of good behaviour, but there are still some drawbacks. In particular, a naive approach can lead to certificates which are very large and for which the proof-checking phase on the code consumer is computationally intensive. There are however techniques which can be used to compress certificates and provide efficient proof-checkers, but again these require considerable effort on the part of the implementer of the PCC system.

We hope to build on experience gained in the earlier Mobile Resource Guarantees project in which a PCC system which could automatically predict and certify memory usage properties of functional programs was developed [15].

4 Prototype Implementation

In order to illustrate the concept of PCC, we have implemented a very basic prototype PCC framework that sits within an OGSA-DAI server. OGSA-DAI (Open Grid Services Architecture - Data Access and Integration) [2] provides a web service interface to existing relational, XML and flat file databases to enable them to be used by remote users in a Grid context. Clients interact with an OGSA-DAI server by uploading *perform documents*. These are XML documents that describe actions to be taken against the database being exposed by the server. For instance, a client may request that a particular SQL query is run against the database, the results converted to CSV, compressed using GZip and sent to a remote FTP server. In OGSA-DAI parlance, each of the individual processes involved is called an *activity*. The perform document describes how activities are plugged together to satisfy the client's request.

At present, the activities present on the server are determined by the server administrator. If a client wants some processing to occur on the
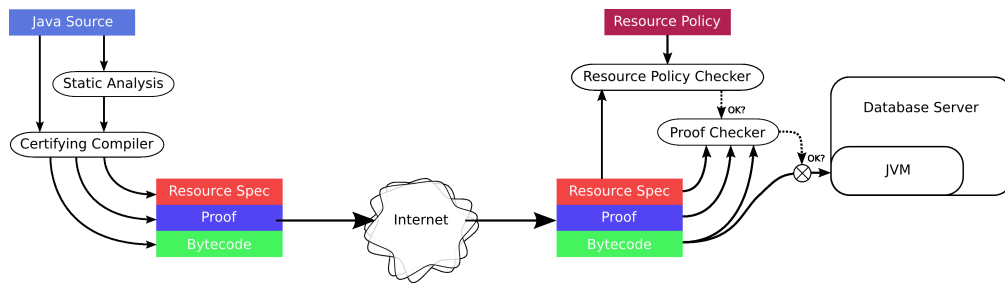

Figure 1: Proof Carrying Code

server, then they must request that this capability be added to the server by its administrators.

We have implemented a new activity, called `MobileCodeActivity`, that allows the client to upload their own Java class in bytecode format to be executed on the server. This code is then integrated into the pipeline constructed by the OGSA-DAI server and used to process data on the server.

4.1 Sending Code

A fragment of a perform document describing an instance of `MobileCodeActivity` is shown in Figure 2. This describes an activity called `mobileCode`, which reads its input from a stream called `input` and outputs to a stream called `results`. These streams can referenced by other activities within the perform document to do other processing on the data. The `classData` element contains the actual Java bytecode that is to be executed, in Base64 encoding. The two attributes of the `classData` element give the class name (required for Java's class loader), and the method to be executed. The rest of the activity description contains the certificate describing the resource usage of the methods in the uploaded class. This is described below.

In our prototype implementation, the method to be executed must have a signature of the form:

```
String methodName (String input)
```

The method must take a string as input, and return a processed string as output. An instantiated `MobileCodeActivity` reads its input, converting it to strings if necessary, processes it using the uploaded code and outputs the result. The library that we use to handle the loading of code and applying arbitrary security checks is described by the third author in [13].

```
<mobileCodeConfig>
 <policy>
  <elem label="replace" count="2"/>
 </policy>

 <externalSpecs>
  <externalSpec
   methodName
    ="java.lang.String.replace">
   <requires>
    <elem label="replace"
        count="1"/>
   </requires>
   <ensures/>
  </externalSpec>
 </externalSpecs>
</mobileCodeConfig>
```

Figure 3: Resource Policy
(set by server administrator)

4.2 Resource Policies

To illustrate PCC in this context, our example uses a simple description of resources in terms of multisets. We use the multiset $\{A, A, B\}$ to specify the a resource consisting of two As and a B. We express multisets in XML by listing each of the resources and their multiplicity, e.g.:

```
<elem label="A" count="2"/>
<elem label="B" count="1"/>
```

We also allow multisets with infinite multiplicity, with the following XML representation:

```
<elem label="A" count="infinity"/>
```

The resource labels are arbitrary strings that have no intrinsic meaning.

The server administrator sets the resource policy of an OGSA-DAI server by writing a configuration file similar to the one shown in Figure 3. This file describes two things.

```
<mobileCode name="mobileCode">
   <mobileCodeInput from="input"/>
   <mobileCodeOutput name="results"/>
   <classData className="TestAgent" methodName="replaceNewlines">
yv66vgAAA... [Java .class file in Base64 encoding]
   </classData>
   [...resource certificate goes here...]
 </mobileCode>
```

Figure 2: Activity Description for Mobile Code

```
<certificate>
 <methodCertificate
   methodName="replaceNewlines">
  <requires>
   <elem label="replace" count="1"/>
  </requires>
  <ensures/>
 </methodCertificate>
 <methodCertificate
   methodName="replaceNewlines2">
  <requires>
   <elem label="replace" count="1"/>
  </requires>
  <ensures/>
  <instructionAssertion offset="2">
   <elem label="replace" count="1"/>
  </instructionAssertion>
 </methodCertificate>
</certificate>
```

Figure 4: Resource Certificate
(uploaded with the code)

Within the `policy` element is the overall re-
source policy that all uploaded code must ad-
here to. In this case, we specify that all
methods are constrained to require at most
two of the "replace" resource. Within
the `externalSpecs` element, we describe the
specification of methods that the uploaded
code is allowed to use. In this case, we
only allow access to one external method,
`java.lang.String.replace`. We specify
that calling this method requires one "replace"
resource (in the `requires` element), and that it
frees up no resources to be used later on (the
empty `ensures` element).

4.3 Resource Certificates

In order to convince the server that the code
adheres to its resource policy, the client must
supply a certificate describing the resource be-
haviour of the code that is uploaded. An ex-
ample such certificate is shown in Figure 4.
Each `methodCertificate` element contains
the certificate for one method. In this exam-
ple, the first method, `replaceNewlines`, re-
quires one "replace" resource and ensures
nothing. This is consistent with the imple-
mentation of the method calling the method
`java.lang.String.replace` once. The sec-
ond method is (spuriously, for the purposes of
this example) more complicated. The original
Java of the body of the method is:

```
for (int i = 0; i < 10; i++) {
 input = input;
}
return input.replace ('\n', ' ');
```

This method obviously does only call
`java.lang.String.replace` once, but
does so after executing a loop. To convince
our checker that this method is safe to run,
we must supply a *loop invariant* that specifies
the property that is always preserved by the
loop. This invariant is supplied in the certificate
in the `instructionAssertion` element.
The instruction at offset 2 in the bytecode is
asserted to require one "replace" resource.
The checker then checks the method by filling
in the resource requirements for every other
instruction in the method.

This process is illustrated in Figure 5. The
left-most column shows the bytecode of the
uploaded method with the certificate provided
by the client. The second column shows
a stage during the completion of the certifi-
cate. The checker starts at the last instruc-
tion of the bytecode. Since this is a re-
turn instruction it takes the stated final re-
source consumption of the method (given by
the `ensures` element in the certificate) and
puts this as a precondition of this instruction.
The checker then works its way backwards

```
0:    iconst_0                    0:    iconst_0                     0:    iconst_0       {replace=1}
1:    istore_1                    1:    istore_1                     1:    istore_1       {replace=1}
2:    iload_1     {replace=1}     2:    iload_1     {replace=1}      2:    iload_1        {replace=1}
3:    bipush 10                   3:    bipush 10                    3:    bipush 10      {replace=1}
5:    if_icmpge 16                5:    if_icmpge 16                 5:    if_icmpge 16   {replace=max(1,1)}
8:    aload_0                     8:    aload_0                      8:    aload_0        {replace=1}
9:    astore_0                    9:    astore_0                     9:    astore_0       {replace=1}
10:   iinc 1, 1                   10:   iinc 1, 1                    10:   iinc 1, 1      {replace=1}
13:   goto 2                      13:   goto 2      {replace=1}      13:   goto 2         {replace=1}
16:   aload_0                     16:   aload_0     {replace=1}      16:   aload_0        {replace=1}
17:   bipush  10                  17:   bipush  10  {replace=1}      17:   bipush  10     {replace=1}
19:   bipush  32                  19:   bipush  32  {replace=1}      19:   bipush  32     {replace=1}
21:   invokevirtual               21:   invokevirtual {replace=1}   21:   invokevirtual {replace=1}
       java/lang/String.replace          java/lang/String.replace          java/lang/String.replace
24:   areturn                     24:   areturn          {}         24:   areturn          {}
```

Figure 5: Checking of certificates

through the instructions. The case of note is the `invokevirtual` instruction that actually invokes the `java.lang.String.replace` method. Here, the checker looks up the specification of the method in the policy and applies it, making the precondition of this instruction the resource {`replace=1`}. The checker then carries on backwards through the code until it gets to the `goto` instruction at offset 13. Here, the next instruction in the control flow is not the next instruction in the method, but is the one at offset 2. So it uses the precondition for the instruction at offset 2 provided by the client to carry on. The checker continues backwards until it gets to offset 5. There is a conditional branch in the code at this point, where the next instruction offset can either be 8 or 16. The checker must combine the resources required by both branches by taking the maximum. When it gets to offset 2, the computed precondition for this offset is compared against the one given by the client. If the given one does not include the computed one, then the method is rejected. The checker then carries on to the top of the method. The computed precondition for the first instruction is the precondition for the whole method. This precondition is checked against the stated precondition for the method, which is in turn checked against the resource policy set by the administrator. If either fail, the code is rejected.

The important point is that the certificate is checked against the code *and* the resource policy. If the checker cannot successfully complete the certificate against the code, then the code is rejected. The client cannot provide an incorrect certificate in an attempt to fool the checker.

4.4 Future Work

The basic system presented here serves to illustrate the idea of PCC but is obviously too basic for actual use. Since the certificate checker only deals with the control flow of the program, and does not take into account the program variables, the resource bounds that are certifiable are very coarse. In order to deal with resources consumed within loops, the certificate must state that the resources allowed for the loop are infinite. We are currently working on more sophisticated forms of certificate that take into account program variables and allow arithmetic reasoning about resource bounds. We have used multisets as our representation of resources in this example, but for more precise control we must use other representations. We are currently experimenting with the use of substructural logics [14] to represent resources.

In order to guarantee the security of our framework, we are also working on a formalised specification of the Java Virtual Machine in the Coq proof assistant [16], as well as a Coq-certified certificate checker for mobile code. The use of mechanised formal reasoning to build our framework will increase confidence in the security that it provides. We are also building a *certifying compiler* that will generate certificates during the compilation process given appropriately annotated Java source.

5 Conclusions

This paper has described some possible strategies for dealing with problems raised by the use of mobile code in Grid environments. Our approach promises to provide strong guarantees of good behaviour for mobile programs, and we hope to develop tools which will enable our methods to be used routinely by e-Scientists. In particular, we plan to significantly extend the possible certificates that can be provided with mobile code to allow arithmetic reasoning about resources rather than statically fixed amounts.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In Rocco De Nicola, editor, *European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, March 2007. To appear.

[2] M. Antonioletti, M.P. Atkinson, R. Baxter, A. Borley, N.P. Chue Hong, B. Collins, N. Hardman, A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N.W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead. The Design and Implementation of Grid Database Services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17(2–4):357–376, February 2005.

[3] Robert Atkey. Specifying and verifying heap space allocation with JML and ESC/Java2. In *8th Workshop for Formal Techniques for Java-like Programs (FT-fJP2006)*, 2006.

[4] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3):273–291, 2006.

[5] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified Memory Usage Analysis. In *Proc. of 13th International Symposium on Formal Methods (FM'05)*, number 3582 in Lecture Notes in Computer Science, pages 91–106, 2005.

[6] Pieter H. Hartel and Luc Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Comput. Surv.*, 33(4):517–558, 2001.

[7] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

[8] Kenneth MacKenzie and Nicholas Wolverson. Camelot and Grail: resource-aware functional programming on the JVM. In *Trends in Functional Programming*, volume 4. Intellect, 2004.

[9] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining static analysis and profiling for estimating execution times. In Michael Hanus, editor, *PADL*, volume 4354 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2007.

[10] Aloysius K. Mok and Weijiang Yu. TIN-MAN: A resource bound security checking system for mobile code. In *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, pages 178–193, London, UK, 2002. Springer-Verlag.

[11] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Proceedings of the Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997. ACM Press.

[12] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[13] Christopher Paton. Web Services & Mobile Code. Honour's Project, University of Edinburgh, 2007.

[14] Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000.

[15] Donald Sannella, Martin Hofmann, David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Hans-Wolfgang Loidl, Kenneth MacKenzie, Alberto Momigliano, and Olha Shkaravska. Mobile resource guarantees. In *Trends in Functional Programming*, volume 6. Intellect, September 2005.

[16] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. INRIA, 2006.

[17] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.