Proof-carrying bytecode

Stephen Gilmore and Matthew Prowse [1,2]

Laboratory for Foundations of Computer Science
The University of Edinburgh, Edinburgh, Scotland

Abstract

In the Mobile Resource Guarantees project's Proof Carrying Code implementation, `.class` files are associated with Isabelle [11] proof scripts containing proofs of bounds on their resource consumption. By using the tools `gf` and `isabelle` on the consumer-side, it is possible to verify after download, that a piece of code conforms to a particular resource policy specified by the consumer, and prevent execution in the event that it does not. We present here a prototype implementation using certain features of the J2SE 5.0 Platform [12]. The (unmodified) bytecode and its proof are packaged as a JAR file for convenient distribution. The codebase uses *Java agents* providing the `Instrumentation` interface, and implements a custom permission class and Security Manager. The external tools are invoked from within Java. Two system commands `makeMRGjar` and `MRGjava` provide a convenient way of using this implementation.

1 Introduction

The purpose of bytecode representations of software is to provide a compact, transmissible representation of compiled code. This allows a compiled program to remain independent of the architecture on which it was compiled, with the bytecode acting as a transport format for the program. Transporting code from provider to consumer raises many issues about such use. The downloaded code might be accidentally or maliciously harmful; there may be differences in functionality between the compilation site and the execution site; or there may be differences between the performance capabilities at the end-points, leading to the downloaded code failing to function because it violates a resource bound of the client machine.

Some of the above problems of bytecode use are well-addressed by modern, well-engineered virtual machines such as the JVM but others are explicitly not addressed. Bounding the consumption of resources at the execution site is not

[1] Email: `stg@inf.ed.ac.uk`
[2] Email: `mprowse@inf.ed.ac.uk`

considered in the Java security architecture. Denial-of-service attacks and failures induced by the violation of a program resource bound are considered to be outside the scope of influence of Java typing and Java security.

In the *proof-carrying code* (PCC) [10] approach, programs can be coupled with proofs which verify desirable properties of the program. This concept radically changes the trust relationship between code producer and code consumer. A security-conscious consumer of code can now download and use code without needing to trust or believe that the code producer is supplying benign code, free from operational flaws. In a PCC architecture proofs may be very expensive to produce but they are very efficient to check, thereby allowing code producers to couple programs and proofs so that a code consumer may separately re-check the proof before executing the program code.

The Mobile Resource Guarantees (MRG) project [9] combines the above themes, applying PCC techniques to bytecode in order to certify those properties of bytecode programs which are not ensured by the execution environment of the virtual machine. Although the pairing of bytecode programs (intended to be portable) and proof-carrying code (intended to be downloaded) might seem to be a natural one, to the best of our knowledge this is the first time that PCC techniques have been applied to bytecode. In [10] and elsewhere, the technique is applied to assembly language code, for example to develop safe assembly language extensions to ML programs.

The Mobile Resource Guarantees project is working towards developing a framework whereby a piece of mobile code can be transmitted by an untrusted code producer across an insecure network, accompanied by an unforgeable and efficiently-checkable proof script that the run-time behaviour (in terms of resource consumption) will remain within some pre-specified bound. The diagram in Figure 1 shows the components of the intended framework.

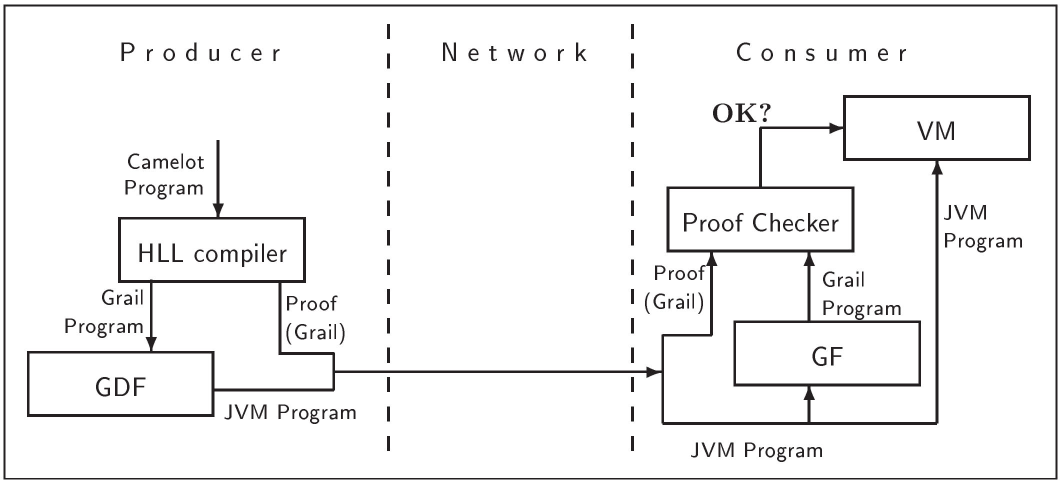

Fig. 1. Mobile Resource Guarantees framework

Our protocol is designed so that it can be integrated with the built-in mechanism for Java bytecode checking, via the *Security Manager*. In the JVM, the Security Manager is entrusted with enforcing the security policy designated by the user, and ensuring that no violations of the security policy occur while the code runs. Our Resource Manager is not a replacement for the standard Java Security Manager but instead forms a *perimeter defense* which prevents certain non-well-behaved programs from being executed at all.

It is well known in security architectures that a perimeter defense alone is frequently inadequate as a complete security solution. For this reason we deploy the Java Security Manager at runtime to trap exceptional cases that are not tracked by our resource management infrastructure. The use of the Java Security Manager as a dynamic regulatory control means that our architecture cannot be less secure than the default Java platform model at runtime, and we believe that our resource management infrastructure does not compromise this security at class load time.

The operational instance in which we work is obtaining certified, structured bytecode via compilation of a high-level impure functional programming language with objects and classes, *Camelot* [13,8,7]. Camelot programs are similar to a dialect of Objective Caml but in contrast to that language which has its own bytecode format and virtual machine, Camelot has neither. Instead Camelot programs are compiled into a structured dialect of Java bytecode called *Grail* [2] and executed on a modern release of the JVM. The Grail language is based on the λJVM language [6] used in the Flint/ML compiler. Grail class files and associated proofs are packaged into JAR archive and downloaded for execution on a configured, PCC-aware instance of Sun's J2SE 5.0 JVM. The specific concern of the present paper is in describing the "back end" of this operation, detailing how the JVM interfaces with the Isabelle theorem prover to check the proof carried with the code.

PCC architectures trade on certificates. Generating the certificate is a time-consuming process performed by the code producer. The source files are compiled to bytecode, and using type inference and possibly additional annotations given by the programmer, the certificate can be generated. The Java class file format has been chosen as a compact, platform independent vehicle for code, and is transmitted together with the certificate, a condensed formal proof in the form of an Isabelle proof script. These are then placed in a JAR file for convenience.

Before executing the bytecode, we wish to verify that runtime behaviour (resource usage) will conform to some *resource policy*, typically that it will run in space linearly bounded by the size of the input. If the check succeeds, we have an absolute guarantee that the resource bounds are met, so it is not necessary to check for resource violations as the code runs. In the event that verification of this property is not possible, execution should be prevented.

The JVM already ensures that untrusted applets execute in a somewhat secured environment (*sandboxing*), but to introduce the necessary runtime checks would be costly in terms of performance. If it is not possible to verify that the code will conform to a given resource policy we might wish to allow execution under close observation. Under these circumstances, instrumenting the code with runtime checks may be a reasonable alternative.

Here we present an implementation of the above framework using certain features of Sun's newest release of the Java Platform, J2SE 5.0. The approach is to instrument the newly obtained code with an additional security check that is performed just as execution begins. We supply our own subclass of the class `java.security.SecurityManager` to handle our new permission, and we can throw an exception in the event of a failed check thus preventing execution.

Structure of this paper:

In the next section we discuss the design and implementation of the support required by the PCC architecture. Following on from this we detail the verification process and discuss JVM configuration. We give an example showing the system in practical use and then give conclusions on the work. The work uses a programming language named Camelot, discussed in the appendix.

2 Design and Implementation

The target platform of the mobile code is the Java Virtual Machine. There are many implementations of JVMs for different systems, and some are more restrictive than others in terms of the language features and libraries they implement. In building this prototype, it has been our aim to investigate a number of the new (and existing) features of Sun's latest Java Platform Standard Edition 5.0 that may be applicable in our context. Although some choices made in the implementation preclude deployment on all JVM platforms, J2SE 5.0 provides a wide range of interesting features.

An advantage of this implementation is that irrespective of their source, class files can be used unmodified. Although we do modify the class as it is loaded, by instrumenting the main method with an additional security check, the file as stored in the filesystem remains unmodified, and the process therefore appears transparent.

The appropriate configuration of the JVM for controlling execution in the manner suggested here requires of few components. Below, we treat each package and class required in turn, describing both its role and how it is used.

2.1 Description of Packages

2.1.1 `mrg.proofchecker`

The `MRGProofChecker` class implements a `check` method which is used to initiate the proof verification process by launching an Isabelle process. The boolean return value of the `check` method depends on the exit status of this process. See the next section for details.

The execution uses the `Runtime.exec` method, which returns a `Process` object. The `waitFor()` method is called on this object requiring the process to terminate, and obtaining an exit value, before continuing execution.

2.1.2 mrg.security

An `MRGPermission` is a named subclass of `BasicPermission`, which has no actions. During the execution of a program, an instance of `MRGPermission` can be used in a call to the `checkPermission` method of the instance of the Java `SecurityManager` which is currently active. If no `SecurityException` is thrown during this call, the deduction which is made is that the code has permission to execute.

The `MRGSecurityManager` class is a subclass of `SecurityManager`, implementing a check for the custom `MRGPermission`. The `checkPermission` method is written to handle `MRGPermission` objects. Permission is granted or denied according to the return value of:

```
MRGProofChecker.check();
```

If this returns `true` then the `checkPermission` method returns quietly and execution will continue, otherwise an instance of `SecurityException` is thrown and execution will terminate if the thrown exception is not subsequently caught.

To use a custom Security Manager, the following command line switch is given to the `java` command:

```
-Djava.security.manager=mrg.security.MRGSecurityManager
```

2.1.3 mrg.javaagent

There are two classes contained within the package `mrg.javaagent`: namely, `mrg.javaagent.MRGAgent` and `mrg.javaagent.MRGTransformer`. The former represents an instance of a "Java agent", a new feature in the J2SE 5.0 platform which uses the latter to *transform* bytecode before it is loaded into memory and initialised. Java agents can be used for several resource-bound-specific purposes [1]:

(i) to query the attached proof and decide to refuse to load, build and execute the class if necessary;

(ii) to apply *per-class* or *per-package* use restrictions by modifying each method in the class with entry and exit assertions that inspect resource consumption measures; and

(iii) to apply *per-method* constraints on heap-allocation and run-time by instrumenting method bodies.

The package `java.lang.instrument` provides services which allow Java agents to instrument programs running on the JVM. The class `MRGAgent` is implemented with a single method, `premain`. When the JVM uses this class as an agent, the `premain` method will be called before the `main` method of the class being executed (hence the name "premain"). To achieve this, a switch of the following form is given to the `java` command:

```
-javaagent:<jarpath>[=<options>]
```

To specify a specific class to be used as an agent, it must be contained within the `.jar` file specified, and the associated metadata file (the "manifest" file) must contain an entry `Premain-Class` which references it.

The signature of the `premain` method is:

```
public static void premain(String arg, Instrumentation i)
```

and it is through using the `Instrumentation` object argument `i` (implementing the `Instrumentation` interface of the newly-added `java.lang.instrument` package) that we can perform bytecode instrumentation. During the `premain` method body, we make a call to:

```
i.addTransformer(new MRGTransformer());
```

This will cause the `transform(...)` method of the new `MRGTransformer` instance to be called once for each subsequent class to be loaded by the JVM.

The class `MRGTransformer` is a bytecode transformer which implements the interface `java.lang.instrument.ClassFileTransformer`. The implementation of the `transform` method uses the code manipulation features of the Byte Code Engineering Library (BCEL) to instrument the class being executed. A security check is prepended to the `main` method before it is constructed in memory and initialised.

An argument of the `transform` method is a byte buffer of the class currently being loaded. If this is the class being executed from the command line, a modified classfile buffer is returned. All other classes that the JVM loads are left unmodified.

Inserting a security check into the `main` method results in the code *self-checking* before execution, ensuring that it does indeed have permission to execute. The security check inserted into the `main` method is bytecode corresponding to the following Java statements:

```
SecurityManager s = System.getSecurityManager();
if (s == null) throw new SecurityException("...");
s.checkPermission(new MRGPermission());
```

To implement this check, we insert a number of new constant pool entries and bytecode instructions to the class and its `main` method. The Byte Code Engineering Library allows us to do this with relatively little programming effort, although it is a fairly heavyweight package, currently a codebase of nearly 500 kilobytes.

3 Performing the Verification

The certificate containing the proof of the program's bounded resource consumption is distributed as an Isabelle `.thy` file. Contained within the `.jar` file, it is first extracted and saved to a temporary file. The proof makes reference to the structure of the program itself, and thanks to the careful construction of the `.class` file, the (functional) Grail form of the program can be reconstructed. This process is called *functionalisation* [3].

By saving to a temporary file the original *.class* file of the program, it can be run through `gf` to obtain an Isabelle `.thy` file containing the Grail abstract syntax. Now these two Isabelle files, along with a precompiled heap file, `VCG`, are sufficient to perform the proof verification.

Invoking Isabelle as a batch process requires a small bootstrap file called `ROOT.ML`. As this file can be read on standard input by the Isabelle process, a temporary file is not needed and the `OutputStream` of the `Process` object corresponding to the Isabelle invocation can be used instead. The input is of the following form:

```
add_path <temp_dir>
use_thy <cert_name>
handle _ => OS.Process.exit OS.Process.failure
```

The way in which the Isabelle process indicates a failed execution of a proof script is by throwing an exception, which is not reflected by the exit status of the process itself. The last line will terminate the process with a failure exit status in the event that such an exception is thrown. This exit status is received by the calling Java code, and a decision to allow execution of the program or not can then be made based upon the success of failure of the proof verification.

4 Launching with `MRGjava`

In order to launch a correctly configured JVM for our purposes we need to link to the implementations of the run-time support for PCC, install the proof-aware resource manager and establish the Java agent to pre-emptively modify the supplied class to invoke the security manager.

4.1 JVM Configuration

First of all the JVM itself needs to be configured with an appropriate classpath, the Java agent, and the security manager. Three switches are given to the `java` process:

- `-cp <path to bcel.jar>:<path to mrg.jar>:<program .jar>`

[3] The tool `gf` is called the *Grail Functionaliser*, whereas the tool `gdf` is the *Grail De-Functionaliser*.

- `-javaagent:<path to mrg.jar>`
- `-Djava.security.manager=mrg.security.MRGSecurityManager`

4.2 Filenames and Paths

The following properties are set using the `-D` option:

- `mrg.jvm.home`
- `mrg.classname`
- `mrg.tmp.directory`

The directory that contains the library JAR files, the executables, and the Isabelle heapfile is passed into the JVM using the `mrg.jvm.home` property. The simple classname of the class executed is set as the value of `mrg.classname`. (This is used as the root of the names of the corresponding certificate and abstract syntax representation of the program, used by Isabelle.) Finally, a directory for temporary files can be specified using the `mrg.tmp.directory` property.

4.3 The `MRGjava` command

The `MRGjava` command conveniently calls `java` setting the appropriate options and properties. The environment variable `MRGJVMHOME` should be set to the directory containing the libraries and executables. This is the value assigned to the `mrg.jvm.home` property for use by the JVM.

The command can take several arguments. The usage format is shown below:

```
MRGjava [-d] [-nocheck] jarFile [arg1 [, arg2 [, ...]]]
```

The only required argument is the path to a JAR file which the user wishes to execute, (provided that `MRGJVMHOME` is set). The `-d` options enable debugging output, and the `-nocheck` option will skip all the code modification and proof checking, simply executing the class directly. Any remaining arguments *after* the JAR file path are passed to the class being executed.

5 Example

The steps involved in going from source code and compilation to proof verification and execution of the insertion sort algorithm are presented here. Two slightly different forms of insertion sort have been used: a simple version which involves fresh allocations on the *free list* [4] at runtime (`InsSort1`), and a version that performs in-place sort (using no additional freelist elements), `InsSort2`.

[4] See the appendix on Camelot for an explanation of the Camelot free list.

At the high-level language level resource types are presented to the application programmer as measurements interleaved in the types which are inferred for the functions of the language. A type is assigned to a Camelot function which includes both the usual Hindley-Milner surface typing specifying the types of the function parameters and result and a cost expression specifying the number of objects allocated in terms of the size of the input data structures. Thus the type system of the Camelot language assigns types to functions which record the number of parameters which they consume, and their types; the type of the result; and the number of diamonds consumed or freed.

The calculation of the object allocation cost of applying a Camelot function is initiated by a syntactic type inference procedure which inspects the abstract structural expression of the function in Damas-Milner style. This produces a set of constraints on heap pre-allocation, consumption and release. These constraints are passed from the Camelot compiler to `lfd_infer` [5], which converts the constraint set into a linear programming problem to be solved by a linear program solver.

The heap-space consumption attributions inferred by the linear program solver are re-expressed as efficiently-checkable assertions in the language of our chosen proof checker.

5.1 Source Code

For simplicity, the definitions of several auxiliary functions have been omitted, such as those used to convert a list of strings to a list of integers. The substance of the Camelot source code of `InsSort1` is as follows.

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

let ins a l =
  match l with
    Nil -> Cons(a,Nil)
  | Cons(x,t) ->
      if a < x then Cons(a,Cons(x,t))
      else Cons(x, ins a t)

let sort l =
  match l with
    Nil -> Nil
  | Cons(a,t) -> ins a (sort t)

let start args =
  let l1 = ( stringList_to_intList args)
  in let _ = print_string ("Input :\n"^(show_list l1)^"\n")
  in let l2 = sort l1
```

```
  in let _   = print_string  ("Result:\n"^(show_list  l2)^"\n")
  in ()
```

The above Camelot program executes as a classical first-order functional program is expected to, with both implicit allocations and deallocations. The original list remains in memory, and the sorted list is a newly created list. When a reference to the original list is no longer retained, it becomes moribund memory to be reclaimed by the garbage collector.

In addition to classical functional programming language constructs Camelot contains language constructs which are rather distinctive, including *destructive matching*. By adding the `@_` modifier to the `Cons` matches in the definitions of `ins` and `sort`, they become destructive matches and the algorithm then operates in-place and eliminates any additional allocations on the free list at runtime. Making these changes to the source of `InsSort2`, the extract below shows the revised definitions:

```
let ins a l =
  match l with
    Nil -> Cons(a,Nil)
  | Cons(x,t)@_ ->
      if a < x then Cons(a,Cons(x,t))
      else Cons(x, ins a t)

let sort l =
  match l with
    Nil -> Nil
  | Cons(a,t)@_ -> ins a (sort t)
```

5.2 Compilation, Certificate Generation and Packaging

Preparing a program for distribution from Camelot source is made very easy using the `makeMRGjar` shell script. Given a source file named `<classname>.cmlt`, the command `makeMRGjar <classname>` is all that is required to compile a program, generate its certificate and package it all in a `.jar` file. Running this command on the `InsSort1` example produces the following output:

```
# makeMRGjar InsSort
Trying to create .jar for InsSort...
Compiled InsSort to JVML
Compiled InsSort$dia_0 to JVML
Successfully created InsSort.jar
#
```

5.3 Execution

To invoke an appropriately configured JVM to check and execute a program with a certificate, the command MRGjava <jarpath> [<args>] performs all the necessary work. Below is shown the result of executing the example:

```
# MRGjava examples/InsSort1.jar 5 4 3 2 1
Input :
[5, 4, 3, 2, 1]
Result:
[1, 2, 3, 4, 5]
#
```

5.4 Failure to Verify

The transcript of execution directly above shows the execution of insertion sort given successful verification of conformance to the resource policy. In the event of unsuccessful verification, the following message appears, offering the user a choice to continue or not.

```
***************************************************************
*** !!! ERROR !!! ERROR !!!                                 ***
***                                                         ***
*** MRGProofChecker failed to verify the given class file.  ***
*** This code may have undesirable effects.                 ***
***                                                         ***
*** To execute the code anyway, respond 'yes' to the        ***
*** following question...                                   ***
***                                                         ***
***************************************************************

Do you wish to execute unverified code:
```

6 Further Work

There may be cases where proof verification fails, such as over-restrictive resource policies on target machines, or transmission errors. Using bytecode instrumentation, one could implement additional runtime checks, at the beginning and/or end of basic blocks of code, ensuring correct entry/exit behaviour. These checks could be carefully chosen to ensure correct operation, at the expense of an increase in processing time, and perhaps memory consumption to some extent too. Properties that could be tested might include ensuring argument list lengths are within agreed bounds, object instantiations are limited to specific portions of the code, etc.

The approach used here of integrating the proof verification with the Java Security Manager falls down when targeting small devices that use J2ME for example. Such platforms do not implement a Security Manager as such, nor can agents be used in the same way. A refinement or alternative approach must be adopted for this to work on such platforms.

7 Conclusions

We have shown that a combination of a modern JVM with a pre-processing phase for bytecode and an independent prover can be used as the foundation of a proof-carrying code infrastructure for bytecode. Programs and proofs can be packaged together using standard file formats such as the Java archive format (JAR). The proofs in our project are specialised ones focussing on resource consumption, especially heap-space consumption of functions as inferred by a resource-aware type system for the high-level object/functional language used as input to the system, Camelot.

The PCC framework rests on relatively heavyweight tools such as the Byte Code Engineering Library and the Isabelle generic theorem prover. The requirement for extra security checking on downloaded code imposes a run-time penalty on programs which could be reduced by tuning our present prototype implementation to include a custom proof-checker component. In particular, we believe that the Isabelle tactics which we use in checking the proof could themselves be improved by using better representations of programs and certificates. Thus the bottleneck is not the use of the Isabelle technology but our tactics which run on top of Isabelle. Such engineering improvements would be necessary to make the promising technology presented here a practical, everyday facility.

Acknowledgements

This research was supported by the Mobile Resource Guarantees (MRG) project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing. The Camelot compiler used in the MRG project was developed by Kenneth MacKenzie and Nicholas Wolverson. The Isabelle tactics and proof scripts used in the present work were developed for the MRG project by David Aspinall, Lennart Beringer and Hans-Wolfgang Loidl. The `gf` and `gdf` tools were developed by Kenneth MacKenzie, using Standard ML code from Peter Bertelsen's SML-JVM.

References

[1] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors,

Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers, volume 3362 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2005.

[2] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In Vladimiro Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

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

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

[5] Steffen Jost. `lfd_infer`: an implementation of a static inference on heap space usage. In *Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2004)*, 2004.

[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] K. MacKenzie and N. Wolverson. Camelot compiler and Grail compiler and decompiler. Archive of Standard ML software available for download at `http://groups.inf.ed.ac.uk/mrg/camelot`.

[8] K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware functional programming for the JVM. In *Trends in Functional Programming*, pages 29–46. Intellect, 2004.

[9] The Mobile Resource Guarantees project. `http://groups.inf.ed.ac.uk/mrg`.

[10] 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.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[12] Sun Microsystems, Inc. Java 2 Platform, Standard Edition 1.5.0, `http://java.sun.com/j2se/1.5.0/`, May 27, 2004.

[13] N. Wolverson and K. MacKenzie. O'Camelot: adding objects to a resource-aware functional language. In *Proceedings of TFP2003*, pages 47–62. Intellect, 2004.

A Performance

With every new technique comes an interest in performing a quantative comparison with existing methods. Figure A.1 shows a comparison of the runtimes of our InsSort2 algorithm using both `java`, and `MRGjava` (without invoking `isabelle`). The overhead incurred by invoking `MRGjava` averages 1 or 2 seconds. The time taken for `isabelle` to perform the proof verification is relatively significant. The current proof for insertion sort takes 12 to 20 seconds to process.

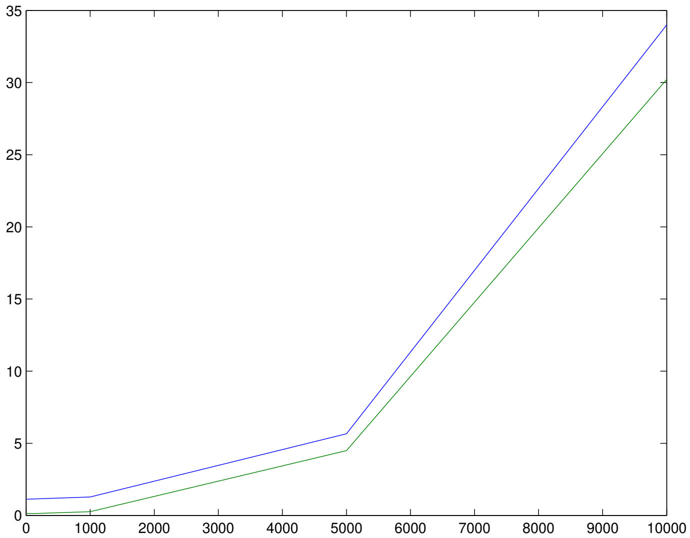

Fig. A.1. Runtimes of `java` and `MRGjava` on `InsSort2` (no proof checking). Runtime in seconds against length of input list.

Although the additional work performed by our augmented JVM is constant, regardless of the size of input, there is the additional layer of processing (the `MRGjava` script itself) whose performance may be related somehow to the number of arguments it receives on the command line. Further work would be necessary here to pin down the cause for the non-constant overhead.

B Camelot

The Camelot compiler targets the Java Virtual Machine and provides language support for reusing memory. The JVM does not provide an instruction to free memory, consigning this to the garbage collector, a generational collector with three generations and implementations of stop-and-copy and mark-sweep collections. The Camelot run-time disposes of unused addresses by adding them to a *free list* of unused memory. On the next allocation caused by the program the storage is retrieved from the head of the free list instead of being allocated by the JVM **new** instruction. When the free list becomes empty the necessary storage is allocated by **new**.

This storage mechanism works for Camelot, but not for Java, because Camelot uses a uniform representation for types which are generated by the compiler, allowing data types to exchange storage cells. This uniform representation is called the *diamond type* [3,4], implemented by a `Diamond` class in the Camelot run-time.

The Camelot compiler is itself implemented in a functional language, Standard ML and has been compiled using the Moscow ML implementation of the language. The compiler is available for free download from the World-Wide Web address http://groups.inf.ed.ac.uk/mrg/camelot/.

B.1 Reuse of addresses

One example of a situation where type-safe reuse of addresses can be used is in a *list updating* function. As with the usual non-destructive list processing, this applies a function to each element of a list in turn, building a list of the images of the elements under the function. In contrast to the usual implementation of a function such as map, the destructive version applies the function *in-place* by overwriting the contents of each cons cell with the image of the element under the function as it traverses the list.

The following simple function increments each integer in an integer list. The Camelot concrete syntax is similar to the concrete syntax of Caml. Where addresses are not manipulated, as here, a Camelot function can also be compiled by Caml.

```
let rec incList lst =
    match lst with
        []      -> []
      | h::t    -> (h + 1) :: incList t
```

This non-destructive version of list processing allocates as many cons-cells as there are elements in the list. With the destructive implementation the storage in the list is reused by overwriting the stored integers with their successors. Thus this version does not allocate any storage.

```
let rec destIncList lst =
    match lst with
        []          -> []
      | (h::t)@d    -> ((h + 1) :: destIncList t)@d
```

In a higher-order version of this function, a *destructive map*, we would have the memory conservation property that if the function parameter does not allocate storage then an application of the destructive map function would not either.

Selective use of in-place update in this way can be used to realise *deforestation*, a program transformation which eliminates unnecessary intermediate data structures which are built as a computation proceeds.

B.2 Affine linear typing of addresses

As an example of a function which is *not* typable in Camelot we can consider the following one. This function attempts to create a modified copy of a list, interleaved with the original list. The (deliberate) error in implementing this function is to attempt to store the cons cells at the front of the list and the cons cell in second place at the same location, `d`.

```
let rec incListCopy lst =
    match lst with
      []         -> []
    | (h::t)@d ->
        let tail = ((h + 1) :: t)@d
        in (h :: tail)@d      (* Error : d used twice! *)
```

This function is faulted by the Camelot compiler with the following diagnostic error message.

```
File "incListCopy.cmlt", line 4-5, characters 18-80:
! ..................let tail = ((h + 1) :: t)@d
!                   in (h :: tail)@d.............
! Variable d of type <> used non-linearly
```

B.3 Control of storage deallocation

The `destIncList` function above demonstrates storage re-use in Camelot. As an example of programmed control of storage deallocation consider the destructive sum function shown below. Summing the elements of an integer list—or more generally folding a function across a list—is sometimes the last operation performed on the list, to derive an accumulated result from the individual values in the list. If that is the case then at this point the storage occupied by the list can be reclaimed and it is convenient to do this while we are traversing the list.

```
let rec destSumList lst =
    match lst with
      []         -> 0
    | (h::t)@_  -> h + destSumList t
```

Matching the location of the object against a wildcard pattern (the _ symbol) indicates that this address is not needed (because it is not bound to a name) and thus it can be freed. The `destSumList` function frees the storage which is occupied by the spine of the list as it traverses the list. In a higher-order version such as *destructive fold* we would have the memory reclamation capability that the function passed in as a parameter could also free the storage occupied by the elements of the list, if these were other storage-occupying objects such as lists or trees.

B.4 Classes and objects

Camelot classes contain methods which can invoke functions which are not associated with any class. At the interface between the object sublanguage and the functional sublanguage of Camelot it is necessary to specify the types of the formal parameters of a function. Thus, in the following (contrived) example it is necessary to specify the type of the parameter of the id function.

```
(* An example Camelot class with a method calling a
   function which is defined outside the class *)
class callExample =
    method myName() : string = id "callExample"
end

let id (s : string) = s
```

As usual, within the functional sublanguage a type inference procedure removes almost all need for the programmer to supply type information.