Mobile Resource Guarantees for Smart Devices*

David Aspinall[1], Stephen Gilmore[1], Martin Hofmann[2], Donald Sannella[1], and Ian Stark[1]

[1] Laboratory for Foundations of Computer Science, School of Informatics, The University of Edinburgh
[2] Lehr- und Forschungseinheit für Theoretische Informatik, Institut für Informatik, Ludwig-Maximilians-Universität München

Abstract. We present the Mobile Resource Guarantees framework: a system for ensuring that downloaded programs are free from run-time violations of resource bounds. Certificates are attached to code in the form of efficiently checkable proofs of resource bounds; in contrast to cryptographic certificates of code origin, these are independent of trust networks. A novel programming language with resource constraints encoded in function types is used to streamline the generation of proofs of resource usage.

1 Introduction

The ability to move code and other active content smoothly between execution sites is a key element of current and future computing platforms. However, it presents huge security challenges — aggravating existing security problems and presenting altogether new ones — which hamper the exploitation of its true potential. Mobile Java applets on the Internet are one obvious example, where developers must choose between sandboxed applets and working within a crippled programming model; or signed applets which undermine portability because of the vast range of access permissions which can be granted or denied at any of the download sites. Another example is open smart cards with multiple applications that can be loaded and updated after the card is issued, where there is currently insufficient confidence in available security measures to take full advantage of the possibilities this provides.

A promising approach to security is *proof-carrying code* [26], whereby mobile code is equipped with independently verifiable certificates describing its security properties, for example type safety or freedom from array-bound overruns. These certificates are condensed and formalised mathematical proofs which are by their very nature self-evident and unforgeable. Arbitrarily complex methods may be used by the *code producer* to construct these certificates, but their verification by the *code consumer* will always be a simple computation. One may compare this to the difference between the difficulty of producing solutions to combinatorial

* This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

problems such as Rubik's cube or satisfiability, and the ease of verifying whether an alleged solution is correct or not.

A major advantage of this approach is that it sidesteps the difficult issue of *trust*: there is no need to trust either the code producer, or a centralized certification authority. If some code comes with a proof that it does not violate a certain security property, and the proof can be verified, then it does not matter whether the code (and/or proof) was written by a Microsoft Certified Professional or a monkey with a typewriter: the property is guaranteed to hold. The user does need to trust certain elements of the infrastructure: the code that checks the proof (although a paranoid user could in principle supply a proof checker himself); the soundness of the logical system in which the proof is expressed; and, of course, the correctness of the implementation of the virtual machine that runs the code — however these components are fixed and so can be checked once and for all. In any case, trust in the integrity of a person or organization is not a reliable basis for trusting that the code they produce contains no undiscovered accidental security bugs! In practice it seems best to take advantage of *both* existing trust infrastructures, which provide a degree of confidence that downloaded code is not malicious and provides desired functionality, *and* the strong guarantees of certain key properties provided by proof-carrying code.

Control of resources (space, time, etc.) is not always recognized as a security concern but in the context of smart cards and other small devices, where computational power and especially memory are very limited, it is a central issue. Scenarios of application which hint at the security implications include the following:

- a provider of distributed computational power may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption;
- third-party software updates for mobile phones, household appliances, or car electronics should come with a guarantee not to set system parameters beyond manufacturer-specified safe limits;
- requiring certificates of specified resource consumption will also help to prevent mobile agents from performing denial of service attacks using bona fide host environments as a portal;

and the one of most relevance in the present context:

- a user of a handheld device, wearable computer, or smart card might want to know that a downloaded application will definitely run within the limited amount of memory available.

The usual way of dealing with programs that exceed resource limits is to monitor their usage and abort execution when limits are exceeded. Apart from the waste that this entails — including the resources consumed by the monitoring itself — it necessitates programming recovery action in the case of failure.

The Mobile Resource Guarantees (MRG) project is applying ideas from proof-carrying code to the problem of resource certification for mobile code. As with other work on proof-carrying code for safety properties, certificates contain formal proofs, but in our case, they claim a resource usage property. Work in MRG has so far concentrated mainly on bounds on heap space usage, but most of the infrastructure that has been built is reusable for bounds on other kinds of resources. One difference between MRG and other work on proof-carrying code is that proof certificates in MRG refer to bytecode programs rather than native code. One bytecode language of particular interest is JVML [22] but there are others, including the CIL bytecode of the Microsoft .NET framework [24], JavaCard [33], and the restricted version of JVML described in [32]. An elegant solution to the tension between the engineering requirement to make theorem proving and proof checking tractable, while at the same time remaining faithful to the imperative semantics of these underlying bytecode languages, is the Grail intermediate language (see Sect. 5) which also targets multiple bytecode languages.

One of the central issues in work on proof-carrying code is how proofs of properties of code are produced. One traditional approach is for object code and proofs to be generated from source code in a high-level language by a *certifying compiler* like Touchstone [10], using types and other high-level source information.[3] The MRG project follows this approach, building on innovative work on linear resource-aware type systems [14,15], whereby programs are certified by virtue of their typing as satisfying certain resource bounds. For instance, in a space-aware type system, the type of an in-place sorting function would be different from the type of a sorting function, like merge sort, that requires extra working space to hold a copy of its input; still different would be the type of a sorting function that requires a specific number of extra cells to do its work, independent of the size of its input. A corresponding proof of this behaviour at the bytecode level can be generated automatically from a typing derivation in such a system in the course of compiling the program to bytecode. It even turns out to be possible to *infer* heap space requirements in many situations [16]. This work has been carried out in a first-order ML-like functional language, Camelot (described in Sect. 3), that has been developed as a testbed by the MRG project. The underlying proof-carrying code infrastructure operates at the bytecode (Grail) level; Camelot is just an example of a language that a code producer might use to produce bytecode together with a proof that it satisfies some desired resource bound.

This paper is an overview of the achievements of the MRG project as of the summer of 2004. It is self-contained, but due to space limitations many points are sketched or glossed over; full technical details can be found in the papers that are cited below. The main contribution of the paper is a presentation of the overall picture into which these technical contributions are meant to fit.

In the next section, we describe the overall architecture of the MRG framework, including the rôle of the two language levels (Grail and Camelot), and how

[3] A slightly different approach was taken by the work on Typed Assembly Language ([25] and later), where a fixed type system is provided for the low-level language, and certification amounts to providing a typing in this low-level type system.

MRG-style proof-carrying code fits with standard Java security. Sections 3 and 4 focus on the "upper" language level, introducing Camelot and space-aware type systems. Section 5 focuses on the "lower" language level, describing the Grail intermediate language and the way that it provides both a tractable basis for proof and relates to (multiple) imperative bytecode languages. Section 6 ties the two language levels together by explaining the logic for expressing proofs of resource properties of bytecode programs and the generation of proofs from resource typings. A conclusion outlines the current status of the MRG project and summarizes its contributions.

2 Architecture and deployment

In this section we discuss the architecture of a smart device-based system which deploys the technology of the MRG project in a novel protocol for certifying resource bounds on downloaded code from an untrusted source. 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.

In our protocol, a *Resource Manager* is responsible for verifying that the certificate supplied with a piece of code ensures that it will execute within the advertised resource constraints. A *Proof Checker* is invoked to do this. 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. 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.

The Mobile Resource Guarantees framework provides a high-level language, Camelot, and a low-level language, Grail, into which this is compiled. (Camelot is presented in more detail in Sect. 3 and Grail is discussed in Sect. 5.) Application developers work in the high-level language and interact with resource typing judgements at the appropriate level of abstraction for their realm of expertise. For this approach to be successful it is necessary for the compilation process to be *transparent* [23] in that the resource predictions made at the high-level language level must survive the compilation process so that they remain true at the low level. This places constraints on the expressive power of the high-level language, prohibiting the inclusion of some more complex language features. It also places constraints on the nature of the compilation process itself, requiring the compiler to sometimes sacrifice peak efficiency for predictability, which is the familiar trade-off from development of real-time software.

A consumer of proof-carrying code (such as Grail class files with attached proofs of resource consumption) requires an implementation technology which enforces the security policy that they specify. The *Java agents* introduced in the J2SDK version 1.5.0 provide the most direct way to implement these policies. An agent is a "hook" in the JVM allowing the PCC consumer to attach their own implementation of their security policy as an instance of a general-purpose PCC Security Manager.

Java agents can be used for several resource-bound-specific purposes:

1. to query the attached proof and decide to refuse to load, build and execute the class if necessary;
2. 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
3. to apply *per-method* constraints on heap-allocation and run-time by instrumenting method bodies.

Each of these checks can be unloaded at JVM instantiation time to allow a mobile-code consumer to vary their security policy between its tightest and laxest extrema.

3 Space types and Camelot

This section describes the high-level language Camelot and the space type system which together allow us to produce JVM bytecode endowed with guaranteed and certified bounds on heap space consumption.

Syntactically, and as far as its functional semantics is concerned, Camelot is essentially a fragment of the ML dialect O'Caml [29]. In particular, it provides the usual recursive datatypes and recursive (not necessarily primitive recursive) definition of functions using pattern matching, albeit restricted to flat patterns.

One difference to O'Caml is that Camelot compiles to JVM bytecode and provides (via the O'Camelot extension [36]) a smooth integration of genuine Java methods and objects.

The most important difference, however, lies in Camelot's memory model. This uses a freelist, managed directly by the compiled code, rather than relying exclusively on garbage collection. All non-primitive types in a Camelot program are compiled to JVM objects of a single class Diamond, which contains appropriate fields to hold data for a single node of any datatype. Unused objects are released to the freelist so that their space can be immediately reused. The compiler generates the necessary code to manage the freelist, based on some language annotations described below.

This conflation of types into a single allocation unit is standard for memory recycling in constrained environments; there is some loss of space around the edges, but management is simple and in our case formally guaranteed to succeed. If required, we could duplicate our analysis to manage a range of cell sizes in parallel, but we have not yet seen compelling examples for this.

3.1 The diamond type

Following [14], Camelot has an abstract type denoted <> whose members are heap addresses of Diamond-objects. The only way to access this type is via datatype constructors. Suppose for example that we have defined a type of integer lists as follows[4]

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

If this is the only type occurring in a program then the Diamond class will look as follows (in simplified form and Java notation):

```
public class Diamond extends java.lang.Object {
    public Diamond R0;
    public int V1;
}
```

If, say, x1 is an element of type iList, hence compiled to an object reference of type Diamond, we can form a new list x2 by

```
let x2 = Cons(9,x1) in ...
```

The required object reference will be taken from the aforementioned freelist providing it is non-empty. Otherwise, the JVM new instruction will be executed to allocate a new object of type Diamond.

If, however, we have in our local context an element d of type $<>$ then we can alternatively form x2 by

```
let x2 = Cons(9,x1)@d in ...
```

thus instructing the compiler to put the new Cons cell into the Diamond object referenced by d, whose contents will be overwritten.

Using these phrases in the context of pattern matching provides us with elements of type $<>$ and also refills the freelist. A pattern match like

```
match x with
   Cons(h,t)@d -> ...
```

is evaluated by binding h, t and d to the contents of the "head" (h) and "tail" (t) fields and the reference to x itself (d). Thus, in the body of the pattern match d is an element of type $<>$ available for constructing new Cons cells.

Alternatively, the syntax

```
match x with
   Cons(h,t)@_ -> ...
```

returns the cell occupied by x to the freelist for later use.

Finally, an unannotated pattern match such as

```
match x with
   Cons(h,t) -> ...
```

performs ordinary non-destructive matching.

---

[4] The annotation ! ensures that the constructor Nil is represented by a null pointer rather than a proper object.

3.2 Linear typing

When a list x is matched against a pattern of the form Cons(h,t)@d or Cons(h,t)@_ it is the responsibility of the programmer to ensure that the list x itself is not used anymore because its contents will be overwritten subsequently. For this purpose, the Camelot compiler has an option that enforces (affine) linear use of all variables. If all variables are used at most once in their scope then there can in particular be no reference to x in the body of the pattern match above. In [14] a formal proof is given that such a program behaves purely functionally, i.e., as if the type $<>$ was replaced by the unit type. Linear typing is, however, a fairly crude discipline and rules out many sound programs. In [6] we present an improved type system that distinguishes between modifying and read-only access to a data structure and in particular allows multiple read-only accesses, which would be ruled out by the linear discipline. This is not yet implemented in Camelot. Alternatively, the programmer can turn off the linear typing option and rely on his or her own judgement, or use some other scheme.

3.3 Extended Example

The code in Figure 1 shows a standalone Camelot application containing a function start : string list -> unit which serves as an entry point. It is assumed that the program is executed by applying start to an (ordinary) list of strings obtained, e.g., from the standard input.

We see that the function ins destroys its argument, whereas the sorting function sort : ilist -> ilist, as well as the display function show_list : ilist -> unit, each leave their argument intact.

3.4 Certification of memory usage

The idea behind certification of heap-space usage in MRG is as follows: given a Camelot program containing a function start : string list -> unit, find a linear function $s(x) = ax + b$ with the property that evaluating (the compiled version of) start on an input list of length n will not invoke the new instruction *provided* that the freelist contains initially no less than $s(n)$ cells.

Once we have such a linear function s we can then package our compiled bytecode together with a wrapper that takes input from stdin or a file, initialises (using new) the freelist to hold $s(n)$ cells where n is the size of the input, and then evaluates start.

3.5 Inference of space bounds

Such linear space bounds can efficiently be obtained using the type-based analysis described in [16] which has subsequently been implemented and tuned to Camelot in [17]. In summary, this analysis infers for each function contained in the program a numerically annotated type describing its space usage. The desired bounding function can then be directly read off from the type of start.

```
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 show_list0 l = match l with
                     Nil -> ""
                   | Cons(h,t) -> begin
                       match t with
                         Nil -> string_of_int h
                       | Cons(h0,t0) -> (string_of_int h) ^ ", " ^ (show_list0 t)
                     end

let show_list l = "[" ^ (show_list0 l) ^ "]"

let stringList_to_intList ss =
        match ss with
          [] -> Nil
        | (h::t) -> Cons((int_of_string h),(stringList_to_intList t))

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

Fig. 1. A standalone Camelot program

```
ins            : 1, int -> iList[0|int,#,0] -> iList[0|int,#,0], 0;
int_of_string  : 0, string -> int, 0;
print_string   : 0, string -> unit, 0;
show_list      : 0, iList [0|int,#,0] -> string, 0;
show_list0     : 0, iList [0|int,#,0] -> string, 0;
sort           : 0, iList [0|int,#,1] -> iList[0|int,#,0], 0;
start          : 0, list_1 [string,#,2|0] -> unit, 0;
stringList_to_intList : 0, list_1 [string,#,2|0] -> iList[0|int,#,1], 0;
string_of_int  : 0, int -> string, 0;
```

Fig. 2. Output of space analysis on the program in Fig. 1

The result of running the analysis on our example program is given in Figure 2. The entry

```
ins : 1, int -> iList[0|int,#,0] -> iList[0|int,#,0], 0;
      ↑
```

indicates that a successful run of ins requires the freelist to contain 1 cell to begin. The entry

```
stringList_to_intList : 0, list_1 [string,#,2|0] -> iList[0|int,#,1], 0;
                                            ↑                      ↑
```

indicates that a call to stringList_to_intList on an input list of length n requires a freelist of size $2n$ and upon completion leaves a freelist of size $1m$ where m is the length of the *resulting* iList.

Finally, the entry

```
start : 0, list_1 [string,#,2|0] -> unit, 0
                            ↑
```

indicates that a call to start requires a freelist of size $2n$ where n is the length of the input, so the desired bounding function can be chosen as $s(n) = 2n$ in this case.

More generally, a (hypothetical) entry

```
f : 3, iList [0|int,#,17] -> iList[0|int,#,13],11
```

would indicate that a call to f : iList -> iList with an argument of length n requires a freelist of minimum size $3 + 17n$ to succeed without invoking new. Moreover, if the resulting list has length m then the freelist will have size at least $11 + 13m$ plus of course the number of cells left over from the initial freelist in case its size was above the minimum specified by the typing.

Actually, the meaning of the constant 17 in the typing is "17 per Cons-cell of the argument" which in the case of linear lists specialises to $17n$ with n being the length. In the case of data structures with more than one constructor, nested data-structures, and tree-like data structures this view is more fine-grained than linear functions in the overall size.

Other examples discussed in [17] include functions on trees such as heap sort and computation of Huffman codes, as well as functions where the space bounding function has fractional coefficients, e.g. $s(n) = \frac{4}{3}n$.

Regarding the functionality of the space inference we note two important aspects. First, the numerical annotations arise as solutions of a system of linear inequalities which is in turn obtained from a skeleton type derivation which has a numerical variable wherever a numerical annotation would be required. Second, the soundness of destructive pattern matches in the sense of Sect. 3.2 arises also as a precondition to the correctness of the space analysis. For further detail we refer to [16].

4 Parameter size

Other guarantees of resource properties that are under consideration in MRG include execution time, stack size, and size of parameters supplied to system calls. Of these three, the third has been studied in more detail albeit not to the same extent as heap space consumption. We summarise the partial results achieved so far.

Suppose we are given a system call

```
brake : int * int -> unit
```

where it is "safety critical" that whenever brake is called with parameters (x,y) then some proposition P(x,y), e.g. a conjunction of linear inequalities describing some "safe window", must be satisfied. We emphasize that the brake function itself will not be implemented in Camelot but assumed as given, for example by being part of the system architecture.

A possible scenario for such a situation could consist of a car manufacturer providing an API which allows for third party firmware updates. The manufacturer would publicise a certain safety policy concerning the parameters supplied in calls to the methods provided in the API. The manufacturer would guarantee that adherence to this safety policy will prevent severe hazards. Within the safety policy the third party provider will try to optimise the behavioural properties of the system. Whether or not such optimisation actually happens need not be formally established; our goal is only to ensure adherence to the manufacturer-supplied safety policy.

In order to express such a safety property on the bytecode level one can use the instrumented operational semantics and bytecode logic which will be described below in Sect. 6.

Here we are concerned with the question of how such safety policies can be expressed on the level of Camelot through a type system in such a way that functions can be checked individually once their typing is known, following the usual typing principles, e.g.: to show that a function has a certain type show that its body has that type assuming it to hold for any recursive call within the body.

We claim that a solution based on *dependent types* in the style of Dependent ML [37] fits these requirements very closely. In a nutshell, the idea is as follows. In order to guarantee that a function main calls brake exclusively with such safe parameters, we may try to type main using dependent types under the following assumed typing for brake. (Since the function brake is not itself implemented in Camelot, we can assume an arbitrary typing for it.)

```
brake : {(x,y) : int * int | P} -> unit
```

We will now explain how this idea has been elaborated within MRG. We re-emphasise that these results are partial as yet and that this section must be understood as a report on on-going work.

4.1 Extending Camelot with dependent types

In order to express the desired typing constraints without overly interfering with the language design, Pfenning and Xi's Dependent ML (DML) [37] appears to be particularly suitable.

DML assumes a simply (i.e. non-dependently) typed base language on top of which dependent types are added in such a way that every dependently typed program also makes sense in the simply typed base language. Moreover, the question whether or not a given simply typed program admits a given dependent typing translates into a constraint solving problem in a certain constraint language over which DML is parametric. For our purposes, we choose linear arithmetic over the integers as constraint language.

DML types may depend on types from the constraint language, but not on other DML types. In particular, the types from the constraint language, the so-called *index sorts*, do not themselves form DML types but can be reflected into DML using singleton types if so desired. Likewise, code may not depend on values of index sorts, only on values of DML types. In this way, all index information may be erased from a DML program and a simply typed program is obtained.

We remark that this is not the case for more radical approaches to dependent typing such as Cayenne [7].

For our purposes, we use linear arithmetic for the constraint language which means that the index sorts include int and are closed under cartesian product. We need two type families: Int, Bool : int −> Type with the intention that Int(i) contains just the integer i and Bool(i) contains true if $1 \leq i$ and false otherwise.

We assume the following constants with typing as indicated:

```
0 : Int(0)
1 : Int(1)
plus : Pi x,y:int.Pi xx:Int(x).Pi yy:Int(y).Int(x+y)
true : Pi x:int|1<=x.Bool(x)
false : Pi x:int|x<=0.Bool(x)
leq : Pi x,y:int.Int(x) -> Int(y) -> Bool(1+y-x)
```

The type former Pi obeys the usual rule for dependent function space due to Martin-Löf: if e : Pi x:t.A(x) and i:t then e[i] : A(i). We use square brackets for dependent application to mark that the argument of such an application is always a term of the constraint language which can be automatically inferred using Xi's elaboration algorithm [37]. Moreover, index application is irrelevant to the actual behaviour of a program; it only affects typeability.

Subset constraints in a Pi-type express that such index arguments must obey a certain constraint and are written using vertical bars as in the typing of true. They can be viewed as syntactic sugar for ordinary dependent application if one closes the index sorts under subsets of index sorts with respect to predicates expressible in the constraint language.

In order to be able to reflect knowledge about branches in a case distinction into the typing we use the following typing rule for if-then-else:

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool}(i) \qquad \Gamma, 1 \leq i \vdash t_2 : T \qquad \Gamma, i \leq 0 \vdash t_3 : T}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T} \quad (\textsc{If})$$

We remark here that typing contexts contain bindings of ordinary variables to types, of index variables to index sorts, and constraints on the index variables.

The remaining types and rules of Camelot remain unchanged. We emphasize that since DML has been developed as an extension to Standard ML and Camelot is equivalent to a subset of Standard ML this extension is unproblematic and not at all innovative. What is innovative is the application of DML to the problem of certifying parameter sizes and also, perhaps, the particular DML signature given above.

4.2 Example

We are now in a position to formally express the constraints on calls to brake as a DML typing.

```
brake : Pi x,y:int|P(x,y).Int(x) -> Int(y) -> Unit
```

For the sake of concreteness assume that P(x,y) is x+y <= 10.

Now suppose that we are given a main function that calls brake with two given parameters but prior to the call checks that these parameters indeed satisfy the required constraint, in order to prevent unsafe calls to brake:

```
brake : Int -> Int -> Unit
main : Int -> Int -> Unit
main = lambda xx:Int.lambda yy:Int.
         if leq(plus(xx,yy),10)
            then brake(xx,yy)
            else brake(0,0)
```

Here is how this program can be typed in our dependently typed extension. As already mentioned the index abstractions and applications can be inferred automatically.

```
brake : Pi x,y:int|x+y<=10.Int(x) -> Int(y) -> Unit
main : Pi x,y:int.Int(x) -> Int(y) -> Unit
main = lambda x,y:int.lambda xx:Int(x).lambda yy:Int(y).
         if leq[x+y,10](plus[x,y](xx,yy),10)
            then brake[x,y](xx,yy)
            else brake[0,0](0,0)
```

Let us see how the typing rule If above allows us to typecheck this definition. Put

$$\Gamma = \mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{int}, \mathsf{xx} : \mathsf{Int}(\mathsf{x}), \mathsf{yy} : \mathsf{Int}(\mathsf{y})$$

and then we have

$$\Gamma \vdash \mathsf{leq}[\mathsf{x}+\mathsf{y}, 10](\mathsf{plus}[\mathsf{x},\mathsf{y}](\mathsf{xx},\mathsf{yy}), 10) : \mathsf{Bool}(1 + 10 - (\mathsf{x}+\mathsf{y}))$$

So, in the "then" branch of the conditional we have the additional constraint

$$1 <= 1+10-(x+y)$$

or equivalently, x+y <= 10, which is what is required to typecheck the application brake[x,y](xx,yy).

4.3 Summary

We have shown in this section how an extension of Camelot with dependent types in the style of DML [37] can be used to automatically check adherence to bounds on parameters to system calls. Of course, this is only a first step. In contrast to the case of heap space certification, we have not yet developed the automatic generation of certificates in bytecode logic for this application. More significantly, generalisations of the described per-call policy will be needed for some applications. For instance, one may want to require some constraint on the parameters supplied to all calls of a given function during a certain interval. As a special case, one might require an upper bound on the number of calls to some function (e.g., network connections) or on the sum of the parameters, e.g., in the case where a system function performs an automatic payment. It remains to be seen to what extent dependent typing provides solutions to these problems as well.

5 Grail

We now move on to the low-level *Grail* language that is the target of the Camelot compiler, and our vehicle for proof-carrying code[5]. The challenge here is to identify a language that not only supports formal proofs of resource usage, based on information from Sect. 3, but also maps directly onto executable bytecodes. To do this we give Grail two distinct semantics, one functional and one imperative. These are provably compatible, and the two viewpoints allow flexible reasoning about resources.

An extended discussion of the properties of Grail appears in [9]. Here we begin by outlining the constraints that shape it. The language for our proof-carrying code needs to be all of the following:

- The target for the Camelot compiler;
- A basis for attaching resource assertions;
- Amenable to formal proof about resource usage;
- The format for sending and receiving guaranteed code;
- Executable.

The first three of these suggest a simple functional language, suitable as the output of a transforming Camelot compiler. This is strengthened by the fact that we must also perform transparent compilation, to preserve resource information computed at the Camelot level. However, the final two requirements demand a ruder machine language: what we guarantee should be the actual resource profile of runnable code.

[5] Grail stands for "Guaranteed Resource Allocation Intermediate Language"

```
method static int fib (int n) =
  let
    val a = 0                               // Local variable declarations
    val b = 1

    fun loop (int a, int b, int n) =        // Local function declaration
      let
        val b = add a b                     // Lexically scoped variables
        val a = sub b a                     // hide outer declarations
        val n = sub n 1
      in
        test(n,a,b)                         // Tail recursive function call
      end

    fun test (int n, int a, int b) =        // Another function declaration
      if n<=1 then b else loop(a,b,n)       // Conditional recursive call
  in
    test(n,a,b)                             // Main expression
  end
```

Fig. 3. Grail code to compute the Fibonacci number F_n. For speed, we keep track of both F_k and F_{k+1} in accumulating parameters a and b.

Our solution is to arrange that Grail programs, such as that in Fig. 3, can be both evaluated functionally, using call-by-value, and executed imperatively, with state and `goto` — with both routes giving exactly the same result. In the functional reading "x=5" is a lexically-scoped declaration; on imperative execution it updates a named storage cell. This dual approach then satisfies all the requirements for our low-level language. The final two requirements are satisfied by providing an assembler from (the imperative interpretation of) Grail to JVM classfiles, which can be executed and transported over a network, and a disassembler that reconstructs the original Grail code.

The functional semantics is comparatively standard: Grail has strong static typing, call-by-value first-order functions, mutually recursive local declarations, and lexical scoping. Within this, we make several simplifications appropriate to a compiler target language. For example, local function declarations may not nest, functions are only applied to values, and expressions can contain just one basic operation; later we shall see some further constraints on control and dataflow.

The Fibonacci code of Fig. 3 is the body of a single method. Above this, Grail provides precisely the class and object structure built into the Java virtual machine. Thus the basic expression operators include not just add and sub but also primitives to create and manipulate objects on the Java heap. We use these to implement the space management inferred for Camelot programs by the analyses of Sect. 3.

The comments in Figure 4 present an alternative view of the same code, as a purely imperative stream of assignment statements and jumps. Instead of

```
method static int fib (int n) =
    let
        val a = 0                           // Initial assignment
        val b = 1                           //    to variables

        fun loop (int a, int b, int n) =    // Labelled basic block, with
            let                             //   live variable annotation
                val b = add a b
                val a = sub b a             // Sequence of assignments
                val n = sub n 1             //   updating named registers
            in
                test(n,a,b)                 // Goto, with live variable
            end                             //   annotation

        fun test (int n, int a, int b) =    // Another labelled basic block
            if n<=1 then b else loop(a,b,n) // Conditional return or jump
    in
        test(n,a,b)                         // Initial entry label
    end
```

Fig. 4. Imperative Grail code to calculate the Fibonacci number F_n. Comments indicate semantics for execution on the Java virtual machine.

local functions we have a collection of basic blocks, function calls are merely jumps, and parameter lists now track which variables are live. This imperative reading gives a direct map onto Java bytecode: Grail variables are JVM variables, and each statement expands to a short sequence of instructions, which compose exactly as laid out in the Grail source. For example:

val b = add a b **val** a = sub b a **val** n = sub n 1	becomes	9 iload_1 10 iload_2 11 iadd 12 istore_2	13 iload_2 14 iload_1 15 isub 16 istore_1	17 iload_0 18 iconst_1 19 isub 20 istore_0

This gives bytecode that is highly stereotyped, and our disassembler recovers the original Grail simply by clustering instruction sequences. We can even identify variable names from standard JVM metadata.

These different views on Grail allow us to support sound formal reasoning, using the logical rules presented in the next section, at the same time as effective transmission and execution, following the architecture of Sect. 2. However, this is only justified if the functional and imperative semantics coincide. We ensure this by placing some additional constraints on Grail. A method declaration is *well-formed* if:

- Local functions are closed (all variables appear in their parameter lists);
- Invocations of local functions are all tail calls;
- The arguments of every function call syntactically match its declared parameters — for example, **fun** f (int x) is always invoked as f(x).

We have a formal semantics for both the functional and imperative views of Grail, defined by induction over the structure of programs. For well-formed code we can prove a strong correspondence between these:

Theorem 1. *Every well-formed Grail method body can be presented either as functional declarations or decomposed into imperative basic blocks:*

$$\mathsf{mbody} \underset{\text{functional}}{\overset{\text{imperative}}{\rightleftarrows}} \mathsf{blocklist}\,.$$

Suppose now that E is a variable environment and s is a matching initial state, appropriate for mbody *and* blocklist *respectively:*

$$E =_{var} s \qquad \textit{where} \qquad var = fv(\mathsf{mbody}) = \mathit{Var}(\mathsf{blocklist})\ .$$

Then functional evaluation and imperative execution coincide: for any final value v

$$E \vdash \mathsf{mbody} \Downarrow_{fun} v \qquad \textit{if and only if} \qquad s \vdash \mathsf{blocklist} \Downarrow_{imp} v\ .$$

Moreover, these evaluations also have identical effect on the heap, and make matching use of time and memory space.

Sect. 6 has more detail on the functional operational semantics, and its accompanying logic. We can apply this theorem to show that evaluation metrics for functional Grail match execution steps of the corresponding imperative Java bytecode [30]. Part of this development includes a formalisation within the Isabelle theorem prover of both functional and imperative semantics, as well as the translation between them.

Further results on the properties of well-formed Grail appear in [9]: relating (functional) free variables to (imperative) liveness; and matching dataflow analysis of imperative single-use registers to a functional linear type system.

To take advantage of these results, our Camelot compiler must of course generate well-formed Grail. To do this it carries out a range of standard transformations, such as λ-lifting, variable renaming, and insertion of intermediate declarations. These are all legitimate functional rearrangements; but in the light of Theorem 1 we can also show that these correspond directly to imperative compilation techniques: namely conversion to static single-assignment form (SSA) and then elimination of Φ-functions [3].

This is an instance of a more general observation, that low-level transformations on registers and imperative variables map to functional transformations of Grail. Thus we can carry out bytecode optimisations like register allocation and sharing while still in the intermediate language of our compiler [35].

Many of the transformations used in compiling Camelot to Grail are familiar from other functional languages; ideas like A-normal and CPS form, types in compilation, and typed low-level languages [2,5,11,12,25,34]. We have taken particular inspiration from λ-JVM, a functional language for expressing general JVM programs [19]. The novelty of Grail, by comparison with these other schemes, lies in the fact that it is strict enough to support a reversible translation to bytecode which preserves execution costs.

Grail generates bytecode with a regular form that makes it particularly straightforward to analyze: for example, the JVM operand stack is always empty between Grail statements, and local variables keep the same type throughout a method body. Simplifications like these appear in other proposals for effective use of Java on smart devices — thus, for example, all Grail programs immediately satisfy Leroy's conditions for fast on-card JavaCard verification [21]. Similarly, the *Squawk* JVM architecture runs on very small devices with a tripartite memory structure (ROM/NVRAM/RAM) [32]: we already satisfy many of the conditions for Squawk bytecode, and we believe that the remaining ones can be ensured by manipulation at the Grail level.

In building a PCC framework with Java classfiles as the transport format, the natural question is: why not just use Java bytecode as the base language? The results presented here give the answer: Grail *is* Java bytecode, but with a stern discipline over the flow of control and data that makes it efficient and straightforward to analyze.

6 Bytecode logic and certificate generation

Our proof-carrying code infrastructure equips Grail programs with *certificates* concerning their resource usage. Certificates contain a claim of resource usage together with (instructions for generating) a proof of the claim. The proof is expressed in a program logic for Grail that we have designed specifically for the purpose. In this section we give an overview of the program logic and then of the process of certificate generation. Full technical details of this work appear in [4,8].

6.1 Resource-counting operational semantics

We want assertions to express properties of program execution as defined by the Grail (functional) operational semantics. The operational semantics is defined as a big-step relation which is annotated with resource measurements. An expression e is evaluated in an environment E and heap h, written

$$E \vdash h, e \Downarrow (h', v, p).$$

to yield a value v, an updated heap h', and a *resource component* p. As usual, an environment is a mapping from variables to values, and a heap is modelled as a finite map from a set of locations to values.

The resource component p is a tuple which includes a measure of the number of instructions executed when evaluating e, and the maximum size of the frame stack. The amount of heap space consumed when evaluating e is not included in p, because it can be calculated as the size of the difference between the domains of the input heap and output heap, $|dom(h') - dom(h)|$. This is possible because we do not model garbage collection; indeed, the JVM specification [22] does not even require garbage collection to occur (and it does not take place on the versions 1.X of the JavaCard platform).

The rules defining the operational semantics are straightforward to write, given knowledge of the translation from Grail into Java bytecode outlined in the previous section. Example rules for `if` statements are shown further below.

6.2 A logic for Grail

A possible starting point for the logic would be to take an existing program logic for Java bytecode, perhaps based on cutting down a logic for Java, and extend it to express the resource-related properties of interest. However, to do this would be to ignore the advantages brought by Grail: rather than attaching assertions to sequences of bytecode instructions, we may attach them to Grail functions, and relate them rather directly to the types used by our Camelot compiler. Moreover, the functional viewpoint afforded by Grail allows more elegant rules in several cases than are possible in a Hoare-style logic.

This led us to design a custom logic of partial correctness for Grail. Sequents are of the form:

$$\Gamma \rhd e : P,$$

relating a Grail expression e to a specification P under some set of assumptions Γ of the same form. The specification P denotes a predicate which constrains possible executions of e as defined by the resource-counting operational semantics.

Satisfaction of a specification P by a program e is denoted by $\models e : P$ and asserts that every (terminating) execution lies within the domain of P, that is

$$\forall E,\ h,\ h',\ v,\ p. \quad E \vdash h, e \Downarrow (h', v, p) \qquad \text{implies} \qquad P(E,\ h,\ h',\ v,\ p).$$

Similarly to VDM, our specification predicates allow us to relate the environment and the initial heap to the result, the final heap and the resources consumed. This means that there is no need for auxiliary variables that are necessary in a Hoare-style logic to relate results in the post-condition to inputs in the pre-condition. This has a particular technical advantage in that we do not require the often rather complicated *adaptation rules* of Hoare logic when using proven (or assumed) specifications for procedures[6].

So far we have not introduced a syntax for writing specification predicates. Instead we use the higher-order logic of the theorem prover, Isabelle/HOL [28], in which we have formalised the entire Grail-based PCC framework. This particular form of shallow embedding for propositions is known as the *extensional* approach. As a rather trivial example of a specification, the predicate $|dom(h)| = |dom(h')|$ is satisfied by programs which do not allocate heap space.

Many of the rules of our logic correspond closely with rules of the operational semantics. For example, the rule for an `if` statement looks like this:

[6] in our case: Grail methods and function calls

$$\frac{\Gamma \rhd e_1 : P_1 \qquad \Gamma \rhd e_2 : P_2}{\begin{array}{l}\Gamma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \lambda\, E\, h\, h'\, v\, p.\, \exists p'.\, p = \mathit{tick}_2(p') \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{true} \longrightarrow P_1(E,\, h,\, h',\, v,\, p')) \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{false} \longrightarrow P_2(E,\, h,\, h',\, v,\, p')) \wedge \\ \qquad\qquad (E\langle x\rangle = \mathsf{true} \vee E\langle x\rangle = \mathsf{false})\end{array}} \quad (\text{IF})$$

In fact, every if statement satisfies a predicate which is equivalent to this form: either the environment binds x to true and we have P_1, or the environment binds x to false and we have P_2. But P_1 and P_2 are not satisfied exactly: we have to adjust the resource component p to account for two extra bytecode instructions (corresponding to the variable lookup and branch).

This rule in the logic captures the behaviour of evaluating either branch of an if statement, expressed in the operational semantics by the two cases:

$$\frac{E\langle x\rangle = \mathsf{true} \quad E \vdash h, e_1 \Downarrow (h_1, v, p)}{E \vdash h, \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow (h_1, v, \mathit{tick}_2(p))} \quad (\text{IF-TRUE})$$

$$\frac{E\langle x\rangle = \mathsf{false} \quad E \vdash h, e_2 \Downarrow (h_1, v, p)}{E \vdash h, \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow (h_1, v, \mathit{tick}_2(p))} \quad (\text{IF-FALSE})$$

which also advance the clock resource component by two steps.

The only place in which the domain of the heap is altered is in Grail's new statement, which corresponds to a `new` statement in Java. This uses a special constructor for a class c which assigns the contents of x_i to each of the n fields z_i in the newly constructed object. The rule in the logic is this:

$$\frac{}{\begin{array}{l}\Gamma \rhd \mathsf{new}\ c\ [\overline{z_i := x_i}] : \lambda\, E\, h\, h'\, v\, p.\ p = \mathit{tick}_{n+1}() \wedge v = \mathsf{Ref}\ \mathit{newloc}(h) \wedge \\ \qquad\qquad h' = h[\mathit{newloc}(h) \mapsto (c, \{\overline{z_i := E\langle x_i\rangle}\})]\end{array}} \quad (\text{VNEW})$$

The function *newloc* models the JVM memory allocator's assignment of a new location which isn't already in the domain of h. An object is modelled as a pair (c, flds) where c is a class name and *flds* is a record assigning field names to values. As usual, the resource component counts the clock ticks: in this case the time taken by a new instruction is $n+1$ ticks. The resulting heap contains a new object with the appropriate fields. This rule captures exactly the behaviour of object construction. Obviously, unrestricted new instructions can lead to uncontrolled growth of the heap. The crux of our memory management and resource assertion system is to severely restrict where new can be used.

Compared with directly expanding operational semantics, the power of the logic comes in the rules for function and method calls. The rules are similar to Hoare's original rule for parameterless procedures (but lacking preconditions, since we have a logic for partial correctness). For a function call, the rule is:

$$\frac{\Gamma, (f(x_1, \ldots, x_n)\ :\ P) \rhd \mathsf{mbody}_f : \lambda\, E\, h\, h'\, v\, p.\, P(E,\, h,\, h',\, v,\, tick_1(call_1(p)))}{\Gamma \rhd f(x_1, \ldots, x_n) : P} \qquad \text{(CALL)}$$

This allows one to recursively use the assumption that a call to f satisfies a specification when proving that the unfolded definition of f (mbody_f) indeed satisfies the specification.[7] Again, however, we must adjust the specification to take account of the resources used in evaluating the function call itself. In this case, the clock advances by one tick and a counter for depth of calls is incremented. In the case of method calls, the depth count corresponds to the number of frames on the framestack; we make a similar count for functions so that we might measure the effects of tail-recursion optimisation. When resource components are combined in let-expressions (corresponding imperatively to sequential composition), the resulting resource component takes the maximum of the depth values in each sub-expression.

As well as the rules corresponding to each syntactic element of Grail, the logic has two essential structural rules:

$$\frac{e : P \in \Gamma}{\Gamma \rhd e : P} \qquad \text{(VAX)}$$

$$\frac{\Gamma \rhd e : P \qquad \forall E\, h\, h'\, v\, p.\ P(E,\, h,\, h',\, v,\, p) \longrightarrow P'(E,\, h,\, h',\, v,\, p)}{\Gamma \rhd e : P'} \qquad \text{(VCONSEQ)}$$

We have established strong results about the bytecode logic, including soundness and (relative) completeness.

Theorem 2. *(Soundness) If $\Gamma \rhd e : P$ then $\Gamma \models e : P$.*

Theorem 3. *(Completeness) If $\models e : P$ then $\rhd e : P$.*

The obvious statement of these theorems belies the complexity of their proofs. A delicate inductive argument on the depth of evaluation and function call nesting is needed to prove the call and method invocation rules sound. To prove completeness, we used a novel technique based on the admissibility of a cut rule for the logic. Full details of the development appear in [4].

[7] Because of the restrictions of Grail described in Sect. 5, the actual parameters $x_1, \ldots, x_n$ coincide with the formal parameters as mentioned in mbody_f. For method calls, however, the appropriate rule must instantiate parameters, substituting into the method body.

6.3 The role of the theorem prover

If we are to believe in the correctness of our approach, it is an essential requirement that the program logic is sound for the semantics of Grail. Plausibility of the whole framework lies with Theorem 2 above.

Taking this point seriously led us to formalise both the program logic and the semantics of Grail within a theorem prover, providing machine-checked proofs of the above theorems. This follows the approach of several other researchers (most closely, Kleymann [18] and Nipkow [27]). In previous work, this methodology was advocated to increase confidence in meta-theoretical results for program logics, especially soundness, to avoid the possibility of embarrassment (experienced by several authors previously) of proposing unsound or inconsistent logics because of subtle flaws in paper-based arguments.[8]

For our PCC infrastructure, this approach provides also the possibility of using the formally derived proof rules to represent proof evidence directly (or indirectly as the result of applications of tactics). This gives us an easy way of constructing certificates, which may be represented simply as proof script texts for Isabelle with a certain format. To check a certificate, we must extract the claim it makes, and see if the proof successfully replays when applied to the code which has been delivered. An advantage of this approach is that the logic is automatically extensible: to satisfy particular resource policies we may draw on additional stock lemmas which amount to derived proof rules in the logic.

The obvious drawback of using Isabelle proof scripts directly is that Isabelle is now required on the client (code consumer) side, and the size of Isabelle's code base and memory footprint precludes its use on most small devices! Of course, one may change the point at which proof-checking is done to be on a securely-connected and trusted *proof server* (employing the strategy known as *off-device verification*), but our viewpoint is that while our present implementation is ideal for an experimental research prototype, it ought to be replaced by a dedicated proof checker for real deployment. A dedicated checker for our logic could be much smaller and more efficient than a general purpose theorem prover.

6.4 Generating certificates

While the bytecode logic outlined above enjoys the property of being relatively complete, our experience is that it is rather too low-level for the straightforward construction of certificates. Our initial strategy was to use (formalised versions of) the space assertions obtained from the space type system as specifications of the corresponding compiled functions. Syntax-directed backwards application of the proof rules for the program logic would then generate purely logical verification conditions arising from side-conditions which should be provable automatically.

[8] Of course, just as paper-based arguments may be scrutinised by many readers, we should encourage at least the *statements* of our formal theorems and the requisite definitions to be examined by others; we may delegate trust in the proofs themselves to the community's trust in the implementation of the theorem prover.

Unfortunately, the hope that this could be achieved turned out to be too naive. Firstly, the generated verification conditions contained many quantifiers, which were not automatically instantiated using Isabelle's standard solvers. More seriously, stronger invariants than just freelist balance were required, in particular invariants concerning separation of certain data structures in the heap (cf. [31]).

Our solution to both problems is to introduce a notion of *derived assertion* which more directly expresses in the logic the semantic intention of notions from the high-level type system described in Sect. 3. These derived assertions do not encompass the full power of the program logic, only that needed to capture the meaning and invariants underlying the space type system.

More concretely, derived assertions have the form

$$e : \{\!|\, \Delta, m \triangleright T, n, U \,|\!\}$$

Here Δ is a typing environment assigning numerically annotated types as in Sect. 3.5 to variables in e, T is a numerically annotated type, and m, n are numbers. Additionally, U records the set of variables that are actually used in e.

A derived assertion expands into an ordinary assertion in the program logic which expresses the semantic meaning of the typing judgement

$$\Delta, m \triangleright e : T, n$$

in the system from [16]. Intuitively, this semantic meaning is as follows: given a stack S and a heap h such that S and h are type-correct with respect to Δ (when Δ says $x : \mathsf{iList}$ then $S[x]$ should indeed point to a linked list in h), then provided e evaluates under S, h to some value v under a purely functional semantics without any space constraints then it will do so in the freelist-based memory model without invoking new provided the freelist has minimum size M. Upon completion the freelist will contain N cells. Here M equals m plus the number of cells obtained from the number of nodes in the data structures pointed to by S according to the numerical annotations in Δ. Likewise, N equals n plus the number of cells obtained from the number of nodes in the data structures pointed to by v according to the numerical annotations in T, plus of course any excess in the initial size of the freelist.

All of this is under the additional assumption that during the evaluation of e no live cell (reachable from the current stack) will be returned to the freelist. As mentioned in Sect. 3.2, this condition is guaranteed under linear typing, and this is currently what is modelled in the derived assertion scheme. That is, in addition to what has been explained already, the derived assertion expresses that the heap regions corresponding to distinct variables listed in U do not overlap.

There are some other technical conditions which turned out to be required. For example, the final heap will equal the initial heap on those locations that are aliased with neither the arguments nor the freelist (i.e. contents of locations not affected by the evaluation should not change).

In total, the definition of the meaning of derived assertions consists of a few hundred lines of Isabelle code. Fortunately, though, we were able to prove once and for all a set of derived proof rules for these derived assertions which roughly follow the typing rules from [16] and allow us to prove derived assertions in a syntax-directed fashion rather than by unfolding definitions. The only non-trivial side-conditions that arise during this syntax-directed backwards application of derived rules are numerical inequalities, all of which turn out to be easily provable provided the derived assertions to start with were constructed from results of the analysis in [17].

Since the analysis [17] speaks about high-level Camelot code whereas the program logic is about compiled Grail, the derived rules sometimes apply to canonical sequences of Grail instructions which arise e.g. from compiling a match construct. It should also be noted that the inference is run on intermediate code in monomorphised A-normal form which is (although syntactically correct Camelot) already quite close to the compiled Grail.

Overall, our approach can be compared to the idea of Foundational Proof-Carrying Code (FPCC) [1], which also takes a formalised machine semantics as a starting point (although for a real machine rather than bytecode), and then derives high-level rules and typing principles. However, whereas FPCC aims at building general derived rules from the ground up, involving complex model constructions, we have instead started from a specific high-level analysis and derived its type soundness directly.

At the time of writing we have conducted very promising experiments (in particular insertion sort and heap sort) where we prove a concrete space bound by first deducing it from an appropriate derived assertion and then proving the latter by backwards application of derived proof rules, where the choice of rule to use is always clear. In principle it is now indeed possible to generate proof scripts automatically during compilation, giving the correct invariants and auxiliary lemmas to be able to establish derived assertions. To make that happen we have designed an Isabelle tactic that can solve derived assertions automatically, given partial typing information from the Camelot compiler. We are now extending this to more examples and connecting to the Camelot compiler to complete the PCC infrastructure. Full details of the certificate generation strategy are given in [8].

7 Conclusions

The MRG project has delivered a prototype framework for guaranteeing resource security in mobile applications, based on proof-carrying code for the Java Virtual Machine. We have demonstrated the feasibility of PCC for resource verification, based on the technologies developed in the project. As part of this, the MRG work has made specific contributions to the relevant state of the art:

– Type systems for memory management in high-level programming languages. These allow static checks on heap usage and automatic inference of space

bounds. For devices with severe memory constraints, this offers the opportunity to raise the current cautious programming model: from manual control of fixed allocation to an automated freelist, without compromising memory safety.
- Resource-exact compilation. Camelot extends the standard task of a compiler — to preserve the meaning of a program — to also reliably preserve resource behaviour. Thus we can use source language types and assertions to correctly describe resource usage for the corresponding executables.
- Grail. Our target language shows not only the practicality of carrying out formal proofs on bytecode; also that a PCC consumer can recover enough structure from the corresponding JVM executable to repeat and verify these proofs.
- The Grail bytecode logic. With its shallow embedding into Isabelle/HOL, this allows us to derive VDM-style assertions of time and space usage for programs. We have formally verified this implementation as sound and complete for a resource-counting operational semantics of Grail.
- The system of derived assertions in the Grail logic. Our logical interpretation of space types provides a toolkit for transferring source-level statements of heap usage into machine-checkable bytecode proofs.
- The MRG architecture. This brings all these components together in an end-to-end PCC framework.

The current system serves as a demonstrator and experimental platform. For practical applications there remain issues of size and performance: although our certificates are small, the trusted code base is large, and programmed for flexibility rather than speed. The present framework, with a full theorem prover at both producer and consumer side, is sufficient for *wholesale* PCC; for example where a software developer passes certified code to a device vendor for approval. A targeted checker built just for the bytecode logic would be considerably smaller, enough to support some retail PCC, where an individual consumer can check downloaded code on their PC before installing on a smart device. Proof-checking on the device itself remains an extremely challenging goal.

Certain components in the MRG framework are natural targets for future development. Automatic certificate generation, as sketched in the last section, has been demonstrated for individual examples, but we need to extend this success to more general settings. *Resource policies* are a user-level description of what a consumer requires of incoming code. We need to investigate how best to express these, and how to map them into specifications in the bytecode logic. Finally, we require a treatment of termination to complement the bytecode logic, which is a logic of partial correctness (all its assertions are contingent on termination). There are established approaches to proving termination; these are in general different to those for correctness, so decoupling them is appropriate, and moreover leads to simpler rules for method and function invocation.

In future MRG work, we look to broaden our programme to address other scenarios for PCC application. These include different kinds of resources, like network connections or concurrent threads; as well as other application domains, such as microcontrollers for embedded systems, or mobile code in the Grid. In this last case, for example, existing systems like the Globus "Resource Specification Language" [13] state hoped-for space and time requirements, possibly even just from back-of-the-envelope calculations, whereas we would aim for static checks of correctness.

At the language level, we propose to transfer some of our work on type systems and logics across to Java itself. We would do this by expressing resource assertions in the industry standard Java Modeling Language (JML), which already has a certain amount of formal tool support [20].

The continuing progress of the project will be publicized on the MRG website, `http://www.lfcs.ed.ac.uk/mrg`. This carries papers and downloadable software as well as a web-based demonstration.

Acknowledgments: We acknowledge the excellent work of the research assistants and students of the Mobile Resource Guarantees project. Kenneth MacKenzie and Nicholas Wolverson led the implementation work on the Camelot compiler. Kenneth developed the Grail assembler and disassembler tools `gf` and `gdf`; Laura Korte and Matthew Prowse also contributed towards `gf`. Lennart Beringer, Hans-Wolfgang Loidl, Alberto Momigliano, Matthew Prowse and Olha Shkaravska developed theorem proving infrastructure for the formalisation of the bytecode logic and proved results about the logic itself. Michal Konečný worked on the type system for the Camelot language, and Robert Atkey investigated related systems. Steffen Jost implemented the `lfd_infer` tool. Roberto Amadio contributed many useful ideas on resource types.

References

1. A. Appel. Foundational proof-carrying code. In *Proceedings of LICS'01*, pages 247–256. IEEE Computer Society Press, 2001.
2. A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
3. A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr. 1998.
4. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proceedings of the 17th International Conference on Theorem Proving in Higher-Order Logics, (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 34–49. Springer, 2004.
5. D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 31(3–4):261–302, 2003.
6. D. Aspinall and M. Hofmann. Another type system for in-place update. In D. Le Métayer, editor, *Programming Languages and Systems (Proceedings of ESOP 2002)*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.
7. L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
8. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Towards certificate generation for linear heap consumption. In *Proceedings of the ICALP/LICS Workshop on Logics for Resources, Processes, and Programs (LRPP2004)*, 2004.

9. L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. In *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, number 85.1 in Electronic Notes in Theoretical Computer Science. Elsevier, June 2003.
10. C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00), Vancouver, Canada*, 2000.
11. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PDLI '93*, ACM SIGPLAN Notices 28(6), pages 237–247, 1993.
12. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. Retrospective on "The essence of compiling with continuations". In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*. ACM Press, 2003.
13. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the IEEE/IFIP 7th International Workshop on Quality of Service*, 1999.
14. M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
15. M. Hofmann. Linear types and non size-increasing polynomial time computation. *Information and Computation*, 183:57–85, 2003.
16. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, New Orleans, 2003.
17. S. 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.
18. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
19. C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proc. 5th SCI World Multiconference*, Workshop on Intermediate Representation Engineering for the Java Virtual Machine. Internat. Inst. of Informatics and Systemics, July 2001.
20. G. Leavens, R. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, 2000.
21. X. Leroy. Bytecode verification on Java smart cards. *Software Practice & Experience*, 32:319–340, 2002.
22. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Jan. 1997.
23. K. MacKenzie and N. Wolverson. Camelot and Grail: resource-aware functional programming for the JVM. In *Trends in Functional Programming*, volume 4, pages 29–46. Intellect, 2004.
24. Microsoft. Overview of the .NET framework. In *.NET Framework Developer's Guide*. `http://msdn.microsoft.com`.
25. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.
26. G. Necula. Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1997.

27. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.
28. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Jan. 2002.
29. O'Caml Web site. The O'Caml Language. `http://www.ocaml.org`.
30. M. Prowse. Proving Grail resource bounds. University of Edinburgh, May 2003.
31. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.
32. N. Shaylor, D. N. Simon, and W. R. Bush. A Java virtual machine architecture for very small devices. In *Language, Compiler, and Tool Support for Embedded Systems: Proceedings of LCTES '03*, number 38(7) in ACM SIGPLAN Notices, pages 31–41, July 2003.
33. Sun Microsystems. *Java Card 2.2 Platform Specification*, 2003. available online at `http://java.sun.com/products/javacard/specs.html`.
34. M. Wand. Correctness of procedure representations in higher-order assembly language. In *Proc. MFPS '91*, LNCS 298, pages 294–311. Springer, 1992.
35. N. Wolverson. Optimisation and resource bounds in Camelot compilation. Laboratory for Foundations of Computer Science, University of Edinburgh, 2003.
36. N. Wolverson and K. MacKenzie. O'Camelot: Adding objects to a resource aware functional language. In *Trends in Functional Programming*, volume 4, pages 47–62. Intellect, 2004.
37. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.