Mobile Resource Guarantees

Chapter 14

Project Evaluation Paper: Mobile Resource Guarantees

Donald Sannella[1], Martin Hofmann[2], David Aspinall[1], Stephen Gilmore[1], Ian Stark[1], Lennart Beringer[1], Hans-Wolfgang Loidl[2], Kenneth MacKenzie[1], Alberto Momigliano[1], Olha Shkaravska[2]

Abstract: The Mobile Resource Guarantees (MRG) project has developed a proof-carrying-code infrastructure for certifying resource bounds of mobile code. Key components of this infrastructure are a certifying compiler for a high-level language, a hierarchy of program logics, tailored for reasoning about resource consumption, and an embedding of the logics into a theorem prover. In this paper, we give an overview of the project's results, discuss the lessons learnt from it and introduce follow-up work in new projects that will build on these results.

14.1 INTRODUCTION

The Mobile Resource Guarantees (MRG) project was a three year project funded by the EC under the FET proactive initiative on Global Computing. The aim of the MRG project was to *develop an infrastructure needed to endow mobile code with independently verifiable certificates describing its resource behaviour*. These certificates are condensed and formalised mathematical proofs of resource-related properties which are by their very nature self-evident, unforgeable, and independent of trust networks. This "proof-carrying-code" (PCC) approach to security [19] has become increasingly popular in recent years [13, 1, 20].

Typical application scenarios for such an infrastructure include the following.

- A provider of a distributed computational power, for example a node in a computational Grid, may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption.

[1]Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland

[2]Inst. f. Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany

- A user of a handheld device or another embedded system might want to know that a downloaded application will definitely run within the limited amount of memory available.

Our PCC infrastructure combines techniques from several different research areas. Most notably, we present a novel approach to PCC of building a hierarchy of logics and of translating high-level language properties into a specialised program logic (see Section 14.3). This approach combines the idea of minimising the proof infrastructure as promoted by foundational PCC [1] with exploiting high-level program properties in the certificates. The properties are expressed in an extended type system and type inference is used for static program analysis. Thus we combine work on program logics in the automated theorem proving community with type-system-based analyses in the programming language community. We also show how the embedding of this hierarchy of logics into the Isabelle/HOL theorem prover yields an executable formalisation that can be directly used in the infrastructure. Since soundness and completeness between the levels are established within the prover, the specialised logic does not enter the trusted code base.

In the following section we will outline the initial objectives of the project (Section 14.2) and then give an overview of the key techniques used, and newly developed, to meet these objectives. We provide an overview of the design of our proof and software infrastructure (Sections 14.3 and 14.4). We summarise the main results in Section 14.5, and discuss future work which builds on these results.

14.2 PROJECT OBJECTIVES

The objectives outlined in our initial proposal strike a balance between foundational and more applied work. The foundational work develops a proof infrastructure built on type systems and program logics. The applied work creates a software infrastructure in a PCC prototype which covers the entire path of mobile code in a distributed system. A general overview of the project, developed about half-way through the project, is presented in [5].

Objective 1 is the development of a framework in which certificates of resource consumption exist as formal objects. This consists of a cost model and a program logic for an appropriate virtual machine and run time environment.

Objective 2 consists of the development of a notion of formalised and checkable proofs for this logic playing the role of certificates.

Objective 3 is the development of methods for machine generation of such certificates for appropriate high-level code. Type systems are used as an underlying formalism for this endeavour. Since resource related properties of programs are

almost always undecidable, we aim — following common practice — for a conservative approximation: there will be programs for which no certificate can be obtained although they may abide by the desired resource policy.

Objective 4 While proof-like certificates are generally desirable, they may sometimes be infeasible to construct or too large to transmit. We therefore study relaxations based on several rounds of negotiation between supplier and user of code leading to higher and higher confidence that the resource policy is satisfied.

We have fully achieved Objectives 1–3, and we started work on Objective 4, which is now being picked up in follow-up projects (see Section 14.5).

14.3 AN INFRASTRUCTURE FOR RESOURCE CERTIFICATION

Developing an efficient PCC infrastructure is a challenging task, both in terms of foundations and engineering. In this section we present the foundational tools needed in such an infrastructure, in particular high-level type-systems and program logics. In terms of engineering, the main challenges are the size of the certificates, the size of the trusted code base (TCB) and the speed of validation.

14.3.1 Proof Infrastructure

In this section we describe the proof infrastructure for certification of resources. This is based on a *multi-layered logics approach* (shown in Figure 14.1), where all logics are formalised in a proof assistant, and meta-theoretic results of soundness and completeness provide the desired confidence.

As the basis we have the (trusted) *operational semantics* which is extended with general "effects" for encoding the basic security-sensitive operations (for example, heap allocation if the security policy is bounded heap consumption). Judgements in the operational semantics have the form $E \vdash h, e \Downarrow h', v, \rho$, where E maps variables to values, h represents the pre-heap and h' the post-heap, and v is the result value, consuming ρ resources. The foundational PCC approach [1] performs proofs directly on this level thereby reducing the size of the TCB, but thereby increasing the size of the generated proofs considerably. To remedy this situation more recent designs, such as the Open Verifier Framework [12] or Certified Abstract Interpretation [10], add untrusted, but provably sound, components to a foundational PCC design.

On the next level there is a general-purpose *program logic* for partial correctness [2, 3]. Judgements in this logic have the form $\Gamma \rhd \mathsf{e} : A$, where the context Γ maps expressions to assertions, and A, an assertion, is a predicate over the parameters of the operational semantics. The role of the program logic is to serve as a platform on which various higher level logics may be unified. The latter purpose makes logical completeness of the program logic a desirable property, which has hitherto been mostly of meta-theoretic interest. Of course, soundness remains mandatory, as the trustworthiness of any application logic defined at higher levels depends upon it. Our soundness and completeness results establish a strong link

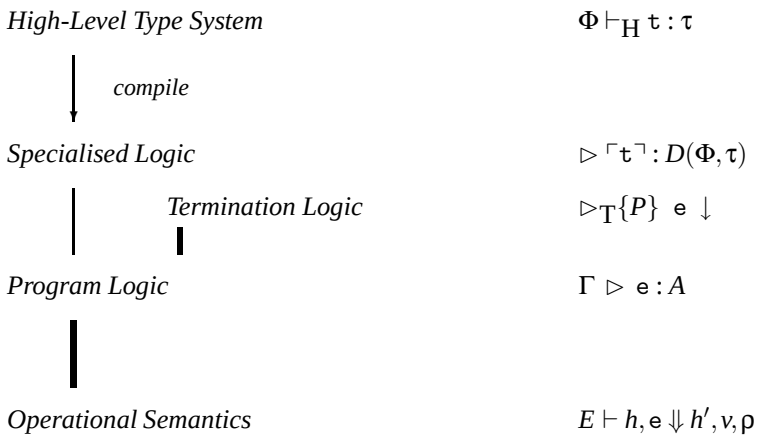


FIGURE 14.1. A family of logics for resource consumption

between operational semantics and program logic, shown as thick lines in Figure 14.1. Note that, since we formalise the entire hierarchy of logics and prove soundness, we do not need to include any of these logics in the TCB.

Whereas assertions in the core logic make statements about partial program correctness, the *termination logic* is defined on top of this level to certify termination. This separation improves modularity in developing these logics, and allows us to use judgements of partial correctness when talking about termination. Judgements in this logic have the form $\triangleright_{\mathrm{T}} \{P\}\ \mathtt{e} \downarrow$, meaning an expression $\mathtt{e}$ terminates under the precondition P.

On top of the general-purpose logic, we define a *specialised logic* (for example the heap logic of [8]) that captures the specifics of a particular security policy. This logic uses a restricted format of assertions, called *derived assertions*, which reflects the judgement of the high-level type system. Judgements in the specialised logic have the form $\triangleright \ulcorner \mathtt{t} \urcorner : D(\Phi, \tau)$, where the expression $\ulcorner \mathtt{t} \urcorner$ is the result of compiling a high-level term $\mathtt{t}$ down to a low-level language, and the information in the high-level type system is encoded in a special form of assertion $D(\Phi, \tau)$ that relies on the context Φ and type τ associated to $\mathtt{t}$. Depending on the property of interest, this level may be further refined into a hierarchy of proof systems, for example if parts of the soundness argument of the specialised assertions can be achieved by different type systems. In contrast to the general-purpose logic, this specialised logic is not expected to be complete, but it should provide support for automated proof search. In the case of the logic for heap consumption, we achieve this by inferring a system of derived assertions whose

level of granularity is roughly similar to the high-level type system. However, the rules are expressed in terms of code fragments in the low-level language. Since the side conditions of the typing rules are computationally easy to validate, automated proof search is supported by the syntax-directedness of the typing rules. At points where syntax-directedness fails — such as recursive program structures — the necessary invariants are provided by the type system.

On the top level we find a *high-level type system* that encodes information on resource consumption. In the judgement $\Phi \vdash_H \mathtt{t} : \tau$, the term $\mathtt{t}$ has an (extended) type τ in a context Φ. This in an example of increasingly complex type systems that have found their way into main-stream programming as a partial answer to the unfeasibility of proving general program correctness. Given this complexity, soundness proofs of the type systems become subtle. As we have seen, our approach towards guaranteeing the absence of bad behaviour at the compiled code level is to translate types into proofs in a suitably specialised program logic.

The case we have worked out in [3] is the Hofmann-Jost type system for heap usage [14] and a simpler instance is given in the rest of this section. In our work, however, we give a general framework for tying such analyses into a fully formalised infrastructure for reasoning about resource consumption.

14.3.2 An Example of a Specialised Program Logic

We now elaborate our approach on a simple static analysis of heap-space consumption based on [11]. The idea is to prove a constant upper bound on heap allocation, by showing that no function allocates heap in a loop. The goal is to detect such non-loop-allocating cases and separate them from the rest, for which no guarantees are given.

It should be emphasised that the heap space analysis in the MRG infrastructure (as shown in Figure 14.5) can handle recursive functions with allocations as long as the consumption can be bounded by a linear function on the input size [14]. We choose this simpler analysis in this section to explain the principles of our approach without adding too much complexity in the logics.

We use the expression fragment of a simple first-order, strict language similar to Camelot [18] (see later in 14.4.1), with lists as the only non-primitive datatype and expressions in administrative-normal-form (ANF), meaning arguments to functions must be variables (k are constants, x variables, f function names):

$$\begin{aligned} e \in \mathit{expr} \quad ::= \quad & k \mid x \mid \mathtt{nil} \mid \mathtt{cons}(x_1, x_2) \mid f(x_1, \ldots, x_{n_f}) \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\ & \mid \mathtt{match}\ x\ \mathtt{with}\ \mathtt{nil} \Rightarrow e_1; \mathtt{cons}(x_1, x_2) \Rightarrow e_2 \end{aligned}$$

We now define a non-standard type system for this language, where $\Sigma(f)$ is a pre-defined type signature mapping function names to $\mathbb{N}$, as follows:

$$\frac{\vdash_H e : n \qquad n \leq m}{\vdash_H e : m}\ (\textsc{Weak}) \qquad \frac{}{\vdash_H k : 0}\ (\textsc{Const}) \qquad \frac{}{\vdash_H x : 0}\ (\textsc{Var})$$

$$\frac{}{\vdash_H f(x_1,\ldots,x_{n_f}) : \Sigma(f)}\ (\textsc{App}) \qquad \frac{}{\vdash_H \texttt{nil} : 0}\ (\textsc{Nil}) \qquad \frac{}{\vdash_H \texttt{cons}(x_1,x_2) : 1}\ (\textsc{Cons})$$

$$\frac{\vdash_H e_1 : m \qquad \vdash_H e_2 : n}{\vdash_H \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : m+n}\ (\textsc{Let}) \qquad \frac{\vdash_H e_1 : n \qquad \vdash_H e_2 : n}{\vdash_H \texttt{match}\ x\ \texttt{with}\ \texttt{nil} \Rightarrow e_1; \texttt{cons}(x_1,x_2) \Rightarrow e_2 : n}\ (\textsc{Match})$$

Let us say that a function is *recursive* if it can be found on a cycle in the call graph. Further, a function *allocates* if its body contains an allocation, i.e, a subexpression of the form $\texttt{cons}(x_1,x_2)$. One can show that a program is typeable iff no recursive function allocates. Moreover, in this case the type of a function bounds the number of allocations it can make.

In order to establish correctness of the type system and, more importantly, to enable generation of certificates as proofs in the program logic, we will now develop a derived assertion and a set of syntax-directed proof rules that mimic the typing rules and permit the automatic translation of any typing derivation into a valid proof.

Recall that $\Gamma \rhd \texttt{e} : A$ is the judgement of the core logic, and that A is parameterised over variable environment, pre- and post-heap (see [2] for more details on encoding program logics for these kinds of languages). Based on this logic, we can now define a *derived assertion*, capturing the fact that the heap h' after the execution is at most n units larger than the heap h before execution[2]:

$$D(n) \equiv \lambda E\ h\ h'\ v\ \rho.\ |dom(h')| \leq |dom(h)| + n$$

We can now prove *derived rules* of the canonical form $\rhd e : D(n)$ to arrive at a program logic for heap consumption:

$$\frac{\rhd e : D(n) \quad n \leq m}{\rhd e : D(m)}\ (\textsc{DWeak}) \qquad \frac{}{\rhd k : D(0)}\ (\textsc{DConst}) \qquad \frac{}{\rhd x : D(0)}\ (\textsc{DVar})$$

$$\frac{}{\rhd f(x_1,\ldots,x_{n_f}) : \Sigma(f)}\ (\textsc{DApp}) \qquad \frac{}{\rhd \texttt{nil} : D(0)}\ (\textsc{DNil}) \qquad \frac{}{\rhd \texttt{cons}(x_1,x_2) : D(1)}\ (\textsc{DCons})$$

$$\frac{\rhd e_1 : D(m) \quad \rhd e_2 : D(n)}{\rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : D(m+n)}\ (\textsc{DLet}) \qquad \frac{\rhd e_1 : D(n) \quad \rhd e_2 : D(n)}{\rhd \texttt{match}\ x\ \texttt{with}\ \texttt{nil} \Rightarrow e_1; \texttt{cons}(x_1,x_2) \Rightarrow e_2 : D(n)}\ (\textsc{DMatch})$$

[2] We do not model garbage collection here, so the size of the heap always increases. This restriction will be lifted in the next section.

We can now automatically construct a proof of bounded heap consumption, by replaying the type derivation for the high-level type system $\vdash_H$, and using the corresponding rules in the derived logic. The verification conditions coming out of this proof will consist only of the inequalities used in the derived logic. No reasoning about the heaps is necessary at all at this level. This has been covered already in the soundness proof of the derived logic w.r.t. the core program logic.

14.3.3 Modelling Reusable Memory

To tackle the issue of reusable memory, we introduce the model of a global "free-list". Heap allocations are fed from the freelist. Furthermore, Camelot provides a destructive pattern `match` operator, which returns the heap cell matched against to the freelist. This high-level memory model is the basis for extending the type system and the logic to a language where memory can be reused.

We can generalise the type system to encompass this situation by assigning a type of the form $\Sigma(f) = (m,n)$ with $m,n \in \mathbb{N}$ to functions and, correspondingly, a typing judgement of the format $\vdash_\Sigma e : (m,n)$. The corresponding derived assertion $D(m,n)$ asserts that if in the pre-heap the global freelist has a length greater than or equal to m, then the freelist in the post-heap has a length greater than or equal to n. Since the freelist, as part of the overall heap, abstracts the system's garbage collection policy, we have the invariant that the size of the post-heap equals the size of the pre-heap.

Now the type of an expression contains an upper bound on the space needed for execution as well as the space left over after execution. If we know that, say, $e : (5,3)$ then we can execute e after filling the freelist with 5 freshly allocated cells, and we will find 3 cells left-over, which can be used in subsequent computations.

The typing rules for this extended system are as follows. Corresponding derived rules are provable in the program logic.

$$\frac{\vdash_H e : (m,n) \quad m' \geq m+q \quad n' \leq n+q}{\vdash_H e : (m',n')}\ (\textsc{Weak}) \qquad \frac{}{\vdash_H k : (0,0)}\ (\textsc{Const}) \qquad \frac{}{\vdash_H x : (0,0)}\ (\textsc{Var})$$

$$\frac{}{\vdash_H f(x_1,\ldots,x_{n_f}) : \Sigma(f)}\ (\textsc{App}) \qquad \frac{}{\vdash_H \texttt{nil} : (0,0)}\ (\textsc{Nil}) \qquad \frac{}{\vdash_H \texttt{cons}(x_1,x_2) : (1,0)}\ (\textsc{Cons})$$

$$\frac{\vdash_H e_1 : (m,n) \quad \vdash_H e_2 : (n,k)}{\vdash_H \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (m,k)}\ (\textsc{Let}) \qquad \frac{\vdash_H e_1 : (m,n) \quad \vdash_H e_2 : (m+1,n)}{\vdash_H \texttt{match}\ x\ \texttt{with}\ \texttt{nil} \Rightarrow e_1; \texttt{cons}(x_1,x_2)@\_ \Rightarrow e_2 : (m,n)}\ (\textsc{Match})$$

Notice that this type system does not prevent deallocation of live cells. Doing so would compromise functional correctness of the code but not the validity of the derived assertions which merely speak about freelist size.

In [8] we extend the type system even further by allowing for input-dependent freelist size using an amortised approach. Here it is crucial to rule out "rogue

programs" that deallocate live data. There are a number of type systems capable of doing precisely that; among them we choose the admittedly rather restrictive linear typing that requires single use of each variable.

14.4 A PCC INFRASTRUCTURE FOR RESOURCES

Having discussed the main principles in the design of the MRG infrastructure, we now elaborate on its main characteristic features (a detailed discussion of the operational semantics and program logic is given in [2]).

14.4.1 Proof Infrastructure

As an instantiation of our multi-layered logics approach, the proof infrastructure realises several program logics, with the higher-level ones tailored to facilitate reasoning about heap-space consumption. While we focus on heap-space consumption here, we have in the meantime extended our approach to cover more general resources in the form of resource algebras [4].

Low-level language: JVM bytecode In order to use the infrastructure in an environment for mobile computation, we focus on a commonplace low-level language: a subset of JVM bytecode. This language abstracts over certain machine-specific details of program execution. Being higher-level than assembler code facilitates the development of a program logic as basis for certification, but also somewhat complicates the cost modelling. For the main resource of interest, heap consumption, allocation is still transparent enough to allow accurate prediction (as shown by the evaluation of our cost model for the JVM). For other resources, in particular execution time, cost modelling is significantly more complicated.

The unstructured nature of JVM code usually gives rise to fairly awkward rules in the operational semantics and in the program logic. We have therefore decided to introduce a slight abstraction over JVM bytecode, *Grail* [9], an intermediate language with a functional flavour, which is in a one-to-one correspondence with JVM bytecode satisfying some mild syntactic conditions. Thus, we can perform certification on the Grail level, and retrieve the Grail code from the transmitted JVM bytecode on the consumer side.

The *operational semantics* for Grail is a resource-aware, big-step semantics over this functional language. Resources are modelled in general terms by specifying a resource algebra over constructs of the language. Separating the rules of the semantics from the propagation of resources makes it easy to model new resources on top of this semantics.

The *program logic* for Grail is a VDM-style partial correctness logic. Thus, it can make meaningful statements about heap consumption, provided that a program terminates. To assure termination, we have also developed a separate termination logic, built on top of the core program logic. It should be emphasised that the program logic does not rely in any way on the Grail code being compiled from a particular high level language. It can be seen as a uniform language for

```
val fac: int -> int -> int
let rec fac n b =
   if n < 1 then b
            else fac (n - 1) (n * b)
```

```
val fac: int -> int
let rec fac n =
  if n < 1 then 1
           else n * fac (n - 1)
```

FIGURE 14.2. Tail-recursive (left) and recursive (right) Camelot code of factorial

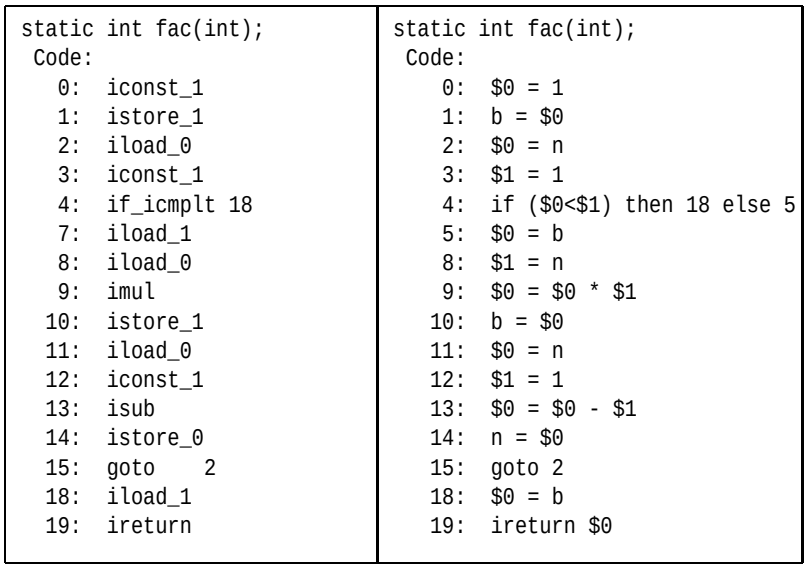
phrasing properties of interest as discussed in the previous section. The benefit of compiling down from a higher-level language is that its additional structure can be used to automatically generate the certificates that prove statements in this program logic.

High-level language: Camelot As high-level language we have defined a variant of OCAML: Camelot [18]. It is a strict functional language with object-oriented extensions and limited support for higher-order functions. Additionally, it has a destructive match statement to model heap deallocation, and it uses a freelist-based heap model that is implemented on top of the JVM's heap model. Most importantly, it is endowed with an inference algorithm for heap-space consumption [14], based on this internal freelist heap model. This inference can derive linear upper bounds for Camelot programs fulfilling certain linearity constraints. Based on this inference, the compiler can also generate a certificate for bounded heap consumption, and it emits a statement in the Grail program logic, expressing this bound for the overall program.

As an example let us examine a tail-recursive and a genuinely recursive Camelot program implementing the factorial function, shown in Figure 14.2. The Java Bytecode corresponding to the tail-recursive Camelot program is given in the first column of Figure 14.3. Recall that many JVM commands refer to the operand stack. If we explicitly denote the items on this stack by `$0, $1, $2,...`, starting from the top, then we obtain a beautified bytecode of the tail-recursive version given in the right column of Figure 14.3. In Grail we take this one step further by removing the stack altogether and allowing arithmetic operations on arbitrary variables. Moreover, we use a functional notation for jumps and local variables as exemplified by the code in the left column of Figure 14.4. In contrast, the genuinely recursive version uses JVM method invocation in the recursive call.

With this functional notation of Grail it is possible to develop a program logic that is significantly simpler compared to other JVM-level logics such as [7]. However, in our work we do not tackle issues such as multi-threading nor do we aim to cover a full high-level language such as Java. We rather focus on the automatic generation of resource certificates.

Meta Logic: Isabelle/HOL In order to realise our infrastructure, we have to select and use a logical framework in the implementation of the hierarchy of pro-



```
static int fac(int);
 Code:
   0:  iconst_1
   1:  istore_1
   2:  iload_0
   3:  iconst_1
   4:  if_icmplt 18
   7:  iload_1
   8:  iload_0
   9:  imul
  10:  istore_1
  11:  iload_0
  12:  iconst_1
  13:  isub
  14:  istore_0
  15:  goto    2
  18:  iload_1
  19:  ireturn
```

```
static int fac(int);
 Code:
    0:  $0 = 1
    1:  b = $0
    2:  $0 = n
    3:  $1 = 1
    4:  if ($0<$1) then 18 else 5
    5:  $0 = b
    8:  $1 = n
    9:  $0 = $0 * $1
   10:  b = $0
   11:  $0 = n
   12:  $1 = 1
   13:  $0 = $0 - $1
   14:  n = $0
   15:  goto 2
   18:  $0 = b
   19:  ireturn $0
```

FIGURE 14.3. Java bytecode in ordinary (left) and beautified (right) form

gram logics. Here we have chosen a very powerful system, Isabelle/HOL, and to definitionally realise the program logic as an inductive definition in the meta logic. To avoid the specification of a separate assertion language, we use a shallow embedding for assertions, which are simply meta-logical predicates over the components of the operational semantics. This simplified approach comes at the expense of an increased trusted code base, since we now have to use an entire instance of Isabelle/HOL in the certificate validation phase, as we will see below. However, we found this choice to be adequate for a prototype system in a scenario of global computing with fairly powerful compute nodes. This choice also enables us to use a very succinct representation of certificates as fragments of Isabelle proof scripts. Even without any semantic compression we achieve a certificate size of about 22-32% of the code size, close to the commonly quoted 20% as an acceptable size for a certificate.

14.4.2 Software Infrastructure

The overall structure of the software infrastructure is depicted in Figure 14.5 and is an instance of a general PCC infrastructure [19] with a code producer (left hand side) and a code consumer (right hand side). The main components on the producer side are a *certifying compiler*, which translates high-level Camelot programs into the Grail intermediate code and additionally generates a certificate of its heap consumption. The latter is formalised as a lemma in the heap space logic for the Grail language [8]. The Grail code is processed by an assembler, the Grail

```
method static int fac (int n) =
 let
   val b = 1
   fun f(int n, int b) =
       if n<1 then b
              else f_else(n,b)

   fun f_else(int n, int b) =
   let
        val b = mul b n
        val n = sub n 1
   in
      f(n,b)
   end
 in
   f(n,b)
 end
```

```
method static int fac (int n) =
 let
   fun f_else(n) =
   let
      val n' = sub n 1
      val n' =
invokestatic <Fac Fac.fac(int)>(n')
   in mul n n'
   end
 in
    if n<1 then 1
           else f_else(n)
 end
```

FIGURE 14.4. Tail-recursive (left) and recursive (right) Grail code of factorial

de-functionaliser (gdf), to generate JVM bytecode. This bytecode is transmitted together with the Isabelle proof script as the certificate of its heap consumption to the code consumer. On the consumer side, the Grail code is retrieved via a disassembler, the Grail functionaliser (gf). Then Isabelle/HOL is used in batch mode to automatically check that the resource property expressed in the attached certificate is indeed fulfilled for this program. Once this has been confirmed the code can be executed on the consumer side.

It should also be noted that the current infrastructure does not represent a closed system, in which all mobile code has to be compiled with the same compiler. While the preferred way of generating a code/certificate pair is to write the program in Camelot and have the compiler automatically produce a certificate, it is also possible to use another high-level language such as Java or Scheme that compiles into JVM bytecode, and to then manually generate a proof for the desired resource property. Since the logic has been formalised in Isabelle/HOL, the entire development infrastructure for this prover is available in generating the certificates. As a mixture of both scenarios, it is also possible to write the top level program in Camelot, and call foreign language code from Camelot. This is particularly useful for accessing Java library functions, e.g. for GUI parts of the code. In [21] an extension of Camelot with object-oriented features is described. These extensions have been used in implementing a directory lookup application to be executed on a PDA, based on the MIDP standard for small devices, which provides a restricted set of Java libraries and is partially based on Sun's KVM.

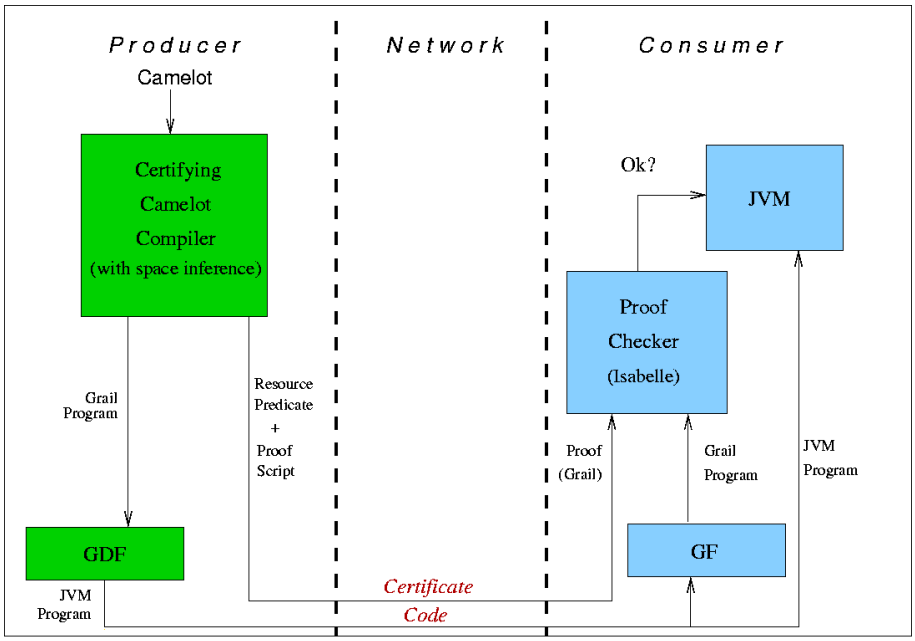


FIGURE 14.5. PCC infrastructure for MRG

14.5 RESULTS

The most visible result of the project is a complete working infrastructure for generating and checking certificates describing the resource behaviour of programs written in a high-level functional programming language. Although the nature of the project was foundational, we emphasised from the start the importance of producing prototypes for the components of the PCC infrastructure — partly as a testbed for experimentation, but also as an on-line test of our techniques in a realistic, distributed setting.

The main novel techniques in the development of the infrastructure are our *multi-layered logics approach* for providing reasoning support tuned to, but not restricted to, the automatic verification of resource properties, and the use of *tactic-based certificates* in order to reduce the size of the certificate, albeit at the cost of increasing the TCB size. However, since we have established soundness of all logics in the prover, of these only the operational semantics needs to be trusted and as validation engine the prover could be replaced by a proof checker with support for a subset of the proof scripting language.

More specifically we have produced the following:

- A *completely formalised virtual machine and cost model* [9] for a JVM-like language. We have used Isabelle/HOL as the theorem proving platform for this formalisation and for encoding the logics.

- A *resource aware program logic* [2, 3] for the bytecode language of the above virtual machine.
- A *specialised logic for heap consumption* [8] that is built on top of the program logic.
- A *certifying compiler* for the strict, first-order functional, object-oriented language Camelot [18], integrated into a prototype PCC infrastructure.
- *Advanced reasoning principles* [14, 17] for resources, based on high-level type systems.

Our particular conclusions on the design of a PCC infrastructure are as follows:

- For automatic certificate generation it is crucial to make use of high-level structural information and to propagate this information down to the program logic. In our design we have realised this as several layers of logics, with the heap logic being tailored to the high-level type-system used to infer information on heap space consumption. In particular, we deliberately depart from the standard approach of splitting certificate validation into verification condition generation and simplification. In our experience, the verification conditions even for simple properties become too complex to be automatically solved by a proof assistant. In contrast, by drawing on information from the high level type inference, we can perform simplifications "on the fly" and thus can keep proofs more manageable.
- The program logic serves as a common language in which to phrase program properties. Thus, program logics over low-level languages can be seen as the "assembler code" for proofs of program properties and as the target language for a compiler that realises high-level type systems to express such properties.
- Encoding the program logic in a proof assistant is not only useful for developing the logic and enforcing formal rigour; it can also serve as an immediate platform for realising the required software infrastructure. While in terms of the size of the TCB and interoperability with other systems a more general format of certificates as proof objects would be favourable, a direct embedding into a proof assistant also yields certificates of small size.
- We found the VDM-style version of the program logic (for partial correctness), with judgements of the form $\Gamma \rhd e : A$, significantly easier to use than an earlier Hoare-style version we had developed, with judgements of the form $\Gamma \rhd \{A\}\ e\ \{A'\}$. This confirms earlier observations on how the need for *auxiliary variables* in a Hoare setting complicates its practical usability [16, 19].

New projects that build on the MRG infrastructure are:

- MOBIUS, an Integrated Project of the FET-GC2 proactive initiative (`http://mobius.inria.fr/`), deals with innovative trust management for global computing, where the resources can be as diverse as network access

and the secure flow of information. In contrast to MRG, this project focuses on Java as a high-level language, and thus will bring the results of our research to a broader community.

- EmBounded, a FET-Open STREP project (`http://www.embounded.org/`), which aims to provide resource bounded computation for embedded systems, using Hume as the high-level programming language. Here we can draw on our amortised costs approach for developing inferences on resource consumption (heap, stack and time) for Hume.
- ReQueST, an EPSRC-funded project (`https://wiki.inf.ed.ac.uk/ReQueST`), aims to develop methods, invent algorithms, and engineer software to equip each request for a Grid service with an irrefutable and accurate certificate which specifies the quantity and type of resources which will be consumed if the request is serviced.

Since the end of MRG, several extensions to the infrastructure as described in this paper have been developed. Related to Objective 4 of the project, on ways of reducing the size of the certificates, we are now studying the use of two forms of resource policies to arrive at a more flexible system without the need of additional communication. In this setup, a guaranteed resource policy is sent together with the certificate. On the consumer side validation of a certificate now involves two steps: a check that the guaranteed resource policy implies the target resource policy on the consumer and validation of the certificate w.r.t. the guaranteed resource policy. Typically, the guaranteed resource policy will contain information about the high-level program, such as the space consumption depending on the input size, and local side-conditions on the consumer are captured in the target resource policy. This approach is discussed in more detail in [6].

Overall we conclude that the project has been very successful in developing the foundations for a novel PCC approach for resources and in producing a prototype infrastructure demonstrating the principles. Finally, visit our project web pages, where you can find project summaries, published papers, and a tutorial [15] with on-line exercises: `http://groups.inf.ed.ac.uk/mrg/`. An on-line demo is directly available at: `http://projects.tcs.ifi.lmu.de/mrg/pcc/`.

ACKNOWLEDGEMENTS

This document summarises work in the MRG project (IST-2001-33149) which was funded by the EC under the FET proactive initiative on Global Computing. We would like to thank the many researchers who contributed to MRG, in particular R. Amadio, R. Atkey, B. Campbell, S. Jost, B. Klin, M. Konečný, M. Prowse, U. Schöpp, and N. Wolverson.

Bibliography

[1] A.W. Appel. Foundational Proof-Carrying Code. In *Symposium on Logic in Computer Science (LICS'01)*, pages 247–258. IEEE Computer Society, June 2001.

[2] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Program Logic for Resource Verification. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, LNCS 3223, pages 34–49. Springer, September 2004.

[3] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Program Logic for Resources. *Theoretical Computer Science*, 2006. Special Issue on Global Computing. To appear.

[4] D. Aspinall, L. Beringer, and A. Momigliano. Optimisation Validation. In *Workshop on Compiler Optimization Meets Compiler Verification (COCV06)*, Vienna, Austria, April 2, 2006. To appear in ENTCS.

[5] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, LNCS 3362, pages 1–26. Springer, 2005.

[6] D. Aspinall and K. MacKenzie. Mobile Resource Guarantees and Policies. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'05)*, LNCS 3956, Nice, March 8–11, 2005. Springer. To appear.

[7] F. Y. Bannwart and P. Müller. A Logic for Bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141(1) of *ENTCS*, pages 255–273. Elsevier, 2005.

[8] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, LNCS 3452, pages 347–362, Montevideo, Uruguay, March 14–18, Feb 2005. Springer.

[9] L. Beringer, K. MacKenzie, and I. Stark. Grail: a Functional Form for Imperative Mobile Code. In *Workshop on Foundations of Global Computing*, volume 85(1) of *ENTCS*. Elsevier, June 2003.

[10] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science. Special Issue on Applied Semantics*, 2006. Also: Tech. Report INRIA-5751. To appear.

[11] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *International Symposium on Formal Methods (FM'05)*, LNCS 3582, pages 91–106, Newcastle, July 18–22, 2005. Springer.

[12] Bor-Yuh Evan Chang, A. Chlipala, G. Necula, and R. Schneck. The Open Verifier Framework for Foundational Verifiers. In *Workshop on Types in Language Design and Implementation (TLDI'05)*. ACM, January 2005.

[13] C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *Conference on Programming Language Design and Implementation (PLDI'00)*, pages 95–107. ACM Press, 2000.

[14] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197, New Orleans, LA, USA, January 2003. ACM Press.

[15] M. Hofmann, H-W. Loidl, and L. Beringer. Certification of Quantitative Properties of Programs. In *Logical Aspects of Secure Computer Systems*, Marktoberdorf, Aug 2-13, 2005. IOS Press. Lecture Notes of the Marktoberdorf Summer School 2005. To appear.

[16] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.

[17] M. Konečný. Functional In-Place Update with Layered Datatype Sharing. In *Intl. Conf. on Typed Lambda Calculi and Applications (TLCA'03)*, LNCS 2701, pages 195–210. Springer, June 2003.

[18] K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware Functional Programming on the JVM. In *Trends in Functional Programing*, volume 4, pages 29–46. Intellect, 2004.

[19] G. Necula. Proof-carrying Code. In *Symposium on Principles of Programming Languages (POPL'97)*, pages 106–116, Paris, France, January 15–17, 1997. ACM Press.

[20] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping Proof Carrying Code. In *Exploring New Frontiers of Theoretical Informatics*, pages 333–347. Kluwer, 2004.

[21] N. Wolverson and K. MacKenzie. O'Camelot: Adding Objects to a Resource Aware Functional Language. In *Trends in Functional Programing*, volume 4, pages 47–62. Intellect, 2004.