

Chapter 1

Mobile Resource Guarantees *Evaluation Paper*

Donald Sannella¹, Martin Hofmann², David Aspinall¹, Stephen Gilmore¹, Ian Stark¹, Lennart Beringer¹, Hans-Wolfgang Loidl², Kenneth MacKenzie¹, Alberto Momigliano¹, Olha Shkaravska²

Abstract: This paper summarises the main outcomes of the Mobile Resource Guarantees (MRG) project, which focused on a proof-carrying-code (PCC) infrastructure for resources to be applied to mobile code. MRG was a three year project funded by the EC under the FET proactive initiative on Global Computing. We give an overview of the projects' results, discuss the lessons learnt from it and introduce follow-up work in new projects that will build on these results.

1.1 INTRODUCTION

The aim of the 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 is the “proof-carrying-code” (PCC) approach to security [18], which has become increasingly popular in recent years [9, 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.

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

²Institut für 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.

In the following section we will outline the initial objectives of the project (Section 1.2) 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 (Section 1.3 and 1.4). We summarise the main results in Section 1.5, and discuss future work which builds on these results.

1.2 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 [4].

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.

At the end of the project we conclude that we have fully achieved Objectives 1–3, and we started work on Objective 4. Topics of Objective 4 that have not yet been addressed will be covered in follow-up projects mentioned in Section 1.5.

1.3 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

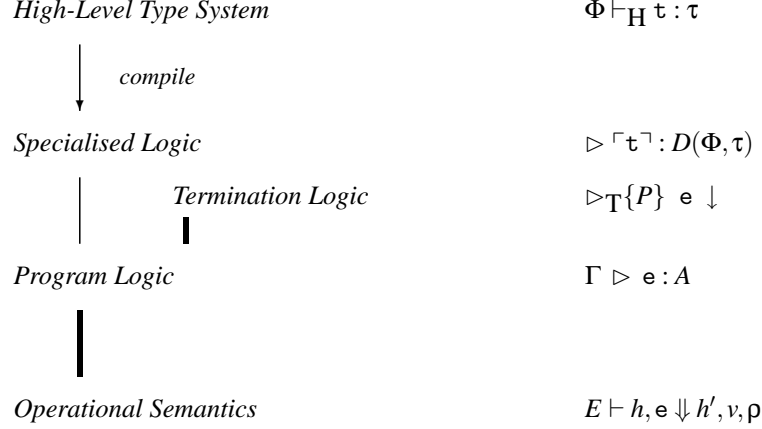


FIGURE 1.1. A family of logics for resource consumption

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 of the certificate.

1.3.1 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 1.1), where all logics are formalised in a proof assistant, and meta-theoretic results of soundness and completeness provide the desired confidence in these components of the trusted code base.

At the basis we have our (trusted) *operational semantics* that is extended with general “effects” which encode 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 $E \vdash h, \mathbf{e} \Downarrow h', v, \rho$, 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 trusted code base.

On the next level there is our general-purpose *program logic* for partial correctness [2, 5]. Judgements in this logic have the form $\Gamma \triangleright \mathbf{e} : A$, 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 pur-

pose 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 between operational semantics and program logic, as shown in Figure 1.1. Note that, since we formalise the entire hierarchy of logics, 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 $\triangleright_T \{P\} e \downarrow$, meaning an expression e terminates under the precondition P .

On top of the general-purpose logic, we define a *specialised logic* (for example the heap logic of [6]) 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 $\triangleright \lceil \tau \rceil : D(\Phi, \tau)$, where the expression $\lceil \tau \rceil$ is the result of compiling a high-level term τ 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 τ . 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 $\Phi \vdash_H \tau : \tau$, the term τ has an (extended) type τ in a context Φ . This is an example of increasingly complex type systems that have found their way into main-stream programming as a partial answer to the un-feasibility 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 [5] is the Hofmann-Jost type system for heap usage [12] 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.

1.3.2 An Example of a Specialised Program Logic

We now elaborate our approach on a simple static analysis of heap-space consumption based on [8]. The idea is to prove a constant upper bound on heap allocation, by showing that no function allocates memory 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. We use a fragment of a simple first-order, strict language similar to Camelot [17] (see later in 1.4.1), with lists as the only non-primitive data-type and expressions in ANF, meaning arguments to functions must be variables (k are constants, x variables, f function names):

$$e \in \text{expr} ::= k \mid x \mid \text{nil} \mid \text{cons}(x_1, x_2) \mid f(x_1, \dots, x_{n_f}) \mid \text{let } x = e_1 \text{ in } e_2 \\ \mid \text{match } x \text{ with nil} \Rightarrow e_1; \text{cons}(x_1, x_2) \Rightarrow e_2$$

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

$$\begin{array}{c} \frac{\vdash_H e : n \quad n \leq m}{\vdash_H e : m} \quad \text{(WEAK)} \qquad \frac{}{\vdash_H k : 0} \quad \text{(CONST)} \qquad \frac{}{\vdash_H x : 0} \quad \text{(VAR)} \\[10pt] \frac{}{\vdash_H f(x_1, \dots, x_{n_f}) : \Sigma(f)} \quad \text{(APP)} \qquad \frac{}{\vdash_H \text{nil} : 0} \quad \text{(NIL)} \qquad \frac{}{\vdash_H \text{cons}(x_1, x_2) : 1} \quad \text{(CONS)} \\[10pt] \frac{\vdash_H e_1 : m \quad \vdash_H e_2 : n}{\vdash_H \text{let } x = e_1 \text{ in } e_2 : m + n} \quad \text{(LET)} \qquad \frac{\vdash_H e_1 : n \quad \vdash_H e_2 : n}{\vdash_H \text{match } x \text{ with nil} \Rightarrow e_1; \text{cons}(x_1, x_2) \Rightarrow e_2 : n} \quad \text{(MATCH)} \end{array}$$

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 $\text{cons}(x_1, x_2)$. 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 our program logic, we will now develop a derived assertion and a set of syntax-directed proof rules that mimic our typing rules and permit the automatic translation of any typing derivation into a valid proof.

Recall that $\Gamma \triangleright e : A$ 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 execution²:

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

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

$$\begin{array}{c}
\frac{\triangleright e : D(n) \quad n \leq m}{\triangleright e : D(m)} \quad (\text{DWEAK}) \qquad \frac{}{\triangleright k : D(0)} \quad (\text{DCONST}) \qquad \frac{}{\triangleright x : D(0)} \quad (\text{DVAR}) \\
\\
\frac{}{\triangleright f(x_1, \dots, x_{n_f}) : \Sigma(f)} \quad (\text{DAPP}) \qquad \frac{}{\triangleright \text{nil} : D(0)} \quad (\text{DNIL}) \qquad \frac{}{\triangleright \text{cons}(x_1, x_2) : D(1)} \quad (\text{DCONS}) \\
\\
\frac{\triangleright e_1 : D(m) \quad \triangleright e_2 : D(n)}{\triangleright \text{let } x = e_1 \text{ in } e_2 : D(m+n)} \quad (\text{DLET}) \qquad \frac{\triangleright e_1 : D(n) \quad \triangleright e_2 : D(n)}{\triangleright \text{match } x \text{ with nil} \Rightarrow e_1; \text{cons}(x_1, x_2) \Rightarrow e_2 : D(n)} \quad (\text{DMATCH})
\end{array}$$

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.

1.3.3 Modelling Reusable Memory

To tackle the issue of reusable memory, we introduce the model of a global “freelist”. Heap allocations are fed from the freelist. Furthermore, our Camelot language 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 our 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.

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

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.

$$\begin{array}{c}
\frac{\vdash_H e : (m, n) \quad m' \geq m + q \quad n' \leq n + q}{\vdash_H e : (m', n')} \quad (\text{WEAK}) \qquad \frac{}{\vdash_H k : (0, 0)} \quad (\text{CONST}) \qquad \frac{}{\vdash_H x : (0, 0)} \quad (\text{VAR}) \\
\\
\frac{}{\vdash_H f(x_1, \dots, x_{n_f}) : \Sigma(f)} \quad (\text{APP}) \qquad \frac{}{\vdash_H \text{nil} : (0, 0)} \quad (\text{NIL}) \qquad \frac{}{\vdash_H \text{cons}(x_1, x_2) : (1, 0)} \quad (\text{CONS}) \\
\\
\frac{\vdash_H e_1 : (m, n) \quad \vdash_H e_2 : (n, k)}{\vdash_H \text{let } x = e_1 \text{ in } e_2 : (m, k)} \quad (\text{LET}) \qquad \frac{\vdash_H e_1 : (m, n) \quad \vdash_H e_2 : (m + 1, n)}{\vdash_H \text{match } x \text{ with nil} \Rightarrow e_1; \text{cons}(x_1, x_2) @ _ \Rightarrow e_2 : (m, n)} \quad (\text{MATCH})
\end{array}$$

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 that merely speak about freelist size.

In [6] 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.

1.4 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]).

1.4.1 Proof Infrastructure

As an instantiation of our multi-layered logic approach, the proof infrastructure realises several program logics, with the higher-level ones tailored to facilitate reasoning about resource consumption. We mainly focus on heap-space consumption here, but have in the meantime extended our approach to cover more general resources in the form of resource algebras [3].

Low-level language: JVM bytecode In order to use the infrastructure in an environment for mobile computation, we focus on a common-place 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 our main resource of interest, heap consumption, allocation is still transparent enough to account for 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. In the follow-up work of the EmBounded project, we therefore deal with lower-level languages down to assembler code.

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

High-level language: Camelot As high-level language we have defined a variant of SML: Camelot. It is a strict functional language with object-oriented extensions and limited support for higher-order functions. Most importantly, it is endowed with an inference algorithm for heap-space consumption which we have presented in [12]. 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.

Meta Logic: Isabelle/HOL In order to realise our infrastructure, we have to select and use a logical framework in the implementation of our hierarchy of program logics. Here we have chosen a very powerful system, Isabelle/HOL, and to definitionally realise the program logic as an inductive definition into the meta

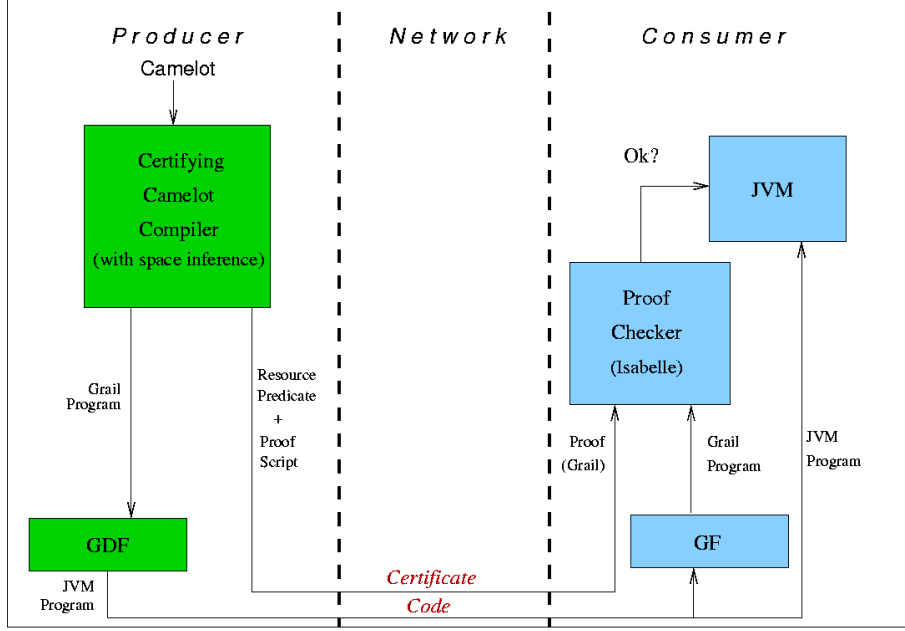


FIGURE 1.2. PCC infrastructure for MRG

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 certificate size of about 22-32% of the code size, close to the commonly quoted 20% as an acceptable size for a certificate.

1.4.2 Software Infrastructure

The overall structure of our software infrastructure is depicted in Figure 1.2 and is an instance of a general PCC infrastructure [18] 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 [6]. The Grail code is processed by an assembler, the Grail

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 the JVM, and to then manually generate a proof for the desired resource property. Since the logic has been formulated 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 other JVM 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. To tackle resource consumption of such mixed code, the foreign function calls can be annotated with their corresponding resource usage, and it becomes possible to analyse and certify the Camelot level heap space consumption of the entire program. Our work on estimating the costs of native methods studies this issue in more detail [10].

A more detailed discussion of the software infrastructure, with worked examples of certificate generation and validation, links to exercises and a discussion of the individual steps sketched here, is presented in [13]. This is a more tutorial style presentation, and will shortly be available in the form of lecture notes for the Marktoberdorf summer school 2005. An on-line version of a demo is available at <http://lionel.tcs.ifi.lmu.de/mrg/pcc4/index.php>, together with a set of exercises.

1.5 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 infra-structure 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, only the operational semantics needs to be trusted. The prover running on the consumer side merely acts as validation engine.

More specifically we have produced the following:

- A *completely formalised virtual machine and cost model* [7] for a JVM-like language. We have used Isabelle/HOL as the theorem proving platform for this formalisation and for encoding the logics on top of it.
- A *resource aware program logic* [2, 5] for the bytecode language of the above virtual machine.
- A *specialised logic for heap consumption* [6] that is built on top of the program logic.
- A *certifying compiler* for the strict, first-order functional, object-oriented language Camelot [17], integrated into a prototype proof-carrying-code infrastructure (available on-line [19]).
- *Advanced reasoning principles* [12, 15] 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 structural information on the high-level 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 to phrase program properties in. 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 the 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 $\Gamma \triangleright e : A$, significantly easier to use than an earlier Hoare-style version we had developed, with judgements of the form $\Gamma \triangleright \{A\} e \{A'\}$. This confirms earlier observations on how the need for *auxiliary variables* in a Hoare setting complicates its practical usability [14, 18].

New projects that build on the MRG infrastructure are:

- EmBounded, a FET-Open STREP project (<http://www.embounded.org/>), which aims to provide resource bounded computation for embedded systems, using Hume [11] as the high-level programming language. Here we can draw on our amortised costs approach for developing inferences on resource consumption for Hume. In particular, we are interested in the heap and stack space consumption, as well as execution time. The presented PCC infrastructure will be the basis for attaching certificates to library functions, ensuring that resource bounded programs can be developed in a compositional way.
- 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.
- 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.

Last, but not least, visit our project web pages, where you can find project summaries, published papers and an interactive demo of the developed infrastructure: <http://groups.inf.ed.ac.uk/mrg/>

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.

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)*, volume 3223 of *LNCS*, pages 34–49, Heidelberg, September 2004. Springer.
- [3] D. Aspinall, L. Beringer, and A Momigliano. Optimisation validation. Technical Report EDI-INF-RR-0509, LFCS, University of Edinburgh, December 2005.
- [4] 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)*, volume 3362 of *LNCS*, pages 1–26. Springer, 2005.
- [5] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. Technical Report EDI-INF-RR-0296, LFCS, University of Edinburgh,, July 2005. Extended version of [2], submitted for journal consideration.
- [6] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, volume 3452 of *LNCS*, pages 347–362, Montevideo, Uruguay, March 14–18, Feb 2005. Springer.
- [7] 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.
- [8] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *International Symposium on Formal Methods (FM'05)*, LNCS. Springer-Verlag, 2005.

- [9] C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *PLDI'00 — Conference on Programming Language Design and Implementation*, pages 95–107. ACM Press, 2000.
- [10] S. Gilmore and O. Shkaravska. Estimating the cost of native method calls for resource-bounded functional programming languages. Submitted to Trends in Functional Programming workshop, February 2005.
- [11] Kevin Hammond and Greg Michaelson. Hume: a domain-specific language for real-time embedded systems. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 37–56, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [12] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, LA, USA, January 2003. ACM Press.
- [13] 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.
- [14] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
- [15] M. Konečný. Functional In-Place Update with Layered Datatype Sharing. In Martin Hofmann, editor, *International Conference on Typed Lambda Calculi and Applications (TLCA'03)*, volume 2701 of *LNCS*, pages 195–210, Heidelberg, June 2003. Springer.
- [16] H-W. Loidl, K. MacKenzie, and S. Jost. A Proof-carrying-code Infrastructure for Resources. In *CC06 — Intl Conf on Compiler Construction*, March 25 – April 2, Vienna, Austria, 2006. Submitted.
- [17] K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware Functional Programming on the JVM. In *Trends in Functional Programming*, volume 4, pages 29–46. Intellect, 2004.
- [18] G. Necula. Proof-carrying Code. In *POPL'97 — Symposium on Principles of Programming Languages*, pages 106–116, Paris, France, January 15–17, 1997. ACM Press.
- [19] D. Sannella and M. Hofmann. Mobile Resource Guarantees. EU OpenFET Project, 2002. <http://groups.inf.ed.ac.uk/mrg/>.
- [20] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping Proof Carrying Code. In J.-J. Levy, E. Mayer, and J. Mitchell, editors, *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.