

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

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

²Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany

Abstract

This paper summarises the main outcomes of the Mobile Resource Guarantees (MRG) project, a three year project funded by the EC under the FET proactive initiative on Global Computing, and discusses follow-up work in new projects that will build on these results.

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” approach to security [12], which has become increasingly popular in recent years [7, 1, 14].

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.
- 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 in the project (Section 2) and then give an overview of the key techniques used, and newly developed, to meet these objectives as well as give an overview of the design of our proving and software infrastructure (Section 3). We summarise the main results in Section 4, and discuss future work that builds on these results.

2 PROJECT OBJECTIVES

The objectives outlined in our initial proposal strike a balance between foundational work on type systems and program logics, to develop a reasoning infrastructure, and more applied work in building a PCC prototype that covers the entire path of mobile code sent in a distributed system, a software infrastructure. A general overview of the project, developed about half-way through the project, is given in [3].

Objective 1 is the development of a framework in which certificates of resource consumption make formal sense. 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 which play 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 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.

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 our *multi-layered logics approach* that drives the development of 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 the validation of the certificate.

3.1 Reasoning infrastructure

In this section we describe the reasoning infrastructure for certification of resources. This is based on a *multi-layered logics approach* (shown in Figure 1), where all layers are formalised in an automated theorem prover, and meta-theoretic results of

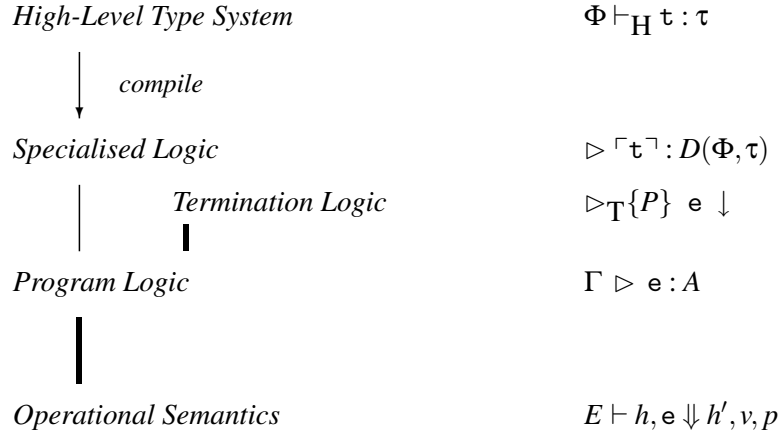


FIGURE 1. A family of logics for resource consumption

soundness and completeness provide the desired confidence in these components of the trusted code base.

We start with an applied type system. While the complexity of proving general program correctness often restricts the research on program verification to only security-critical systems, increasingly complex type systems have found their way into main-stream programming and are accepted as useful tools in software development. Given this complexity, soundness proofs become subtle, and the user of the code has to trust both the proof and the translation of the high-level code into the object-code. Our approach in guaranteeing the absence of bad behaviour is to translate types into proofs in a suitably specialised program logic.

At the basis we have our (trusted) *operational semantics* that is extended with general “effects” that 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 $E \vdash h, e \Downarrow h', v, p$, where E maps variables to values, h represents the pre-heap and h' the post-heap, and v is the result value, consuming p resources. The Foundational PCC approach [1] performs proofs directly on this level and thereby reduces the size of the trusted code base (TCB). Note that, since we formalise the entire hierarchy of logics in a theorem prover, we do not need to include any of these logics into the TCB either.

On the next level we have our general-purpose *program logic* for partial correctness. Judgements in this logic have the form $\Gamma \triangleright e : A$, where the context Γ maps expressions to assertions, and A , an assertion, is a predicate over the parameters to the operational semantics. The role of the program logic is to serve as a platform at 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 tight link between operational semantics and program logic, as shown in Figure 1.

While assertions in the core logic make statements on 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 the partial correctness logic when talking about termination. Judgements in the termination logic have the form $\triangleright_{\top}\{P\} \ e \ \downarrow$, meaning an expression e terminates under the precondition P .

On top of the general-purpose logic, a *specialised logic* (for example the heap logic of [4]) is defined that captures the specifics of a particular security policy. This logic uses a restricted format of assertions, called *derived assertions*, which reflects the information of the high-level type system. Therefore, the specialised logic can be embedded into the core logic. 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 that depends on the context Φ and type τ associated to τ , $D(\Phi, \tau)$. 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 logically complete, but it should provide support for automated proof search. In the case of the logic for heap consumption, this is achieved by formulating 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-directness of the typing rules. At points where syntax-directness 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 level is of immediate relevance for the programming languages area, and many type-based inferences have been suggested. The case we have worked out in [2] is the Hofmann & Jost type system for heap usage [9] 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.

We now elaborate our approach on a simple static analysis of heap-space consumption based on the work by Cachera and Jensen [6]. The idea is to prove a constant upper bound on heap allocation, by proving that no function allocates memory in a loop. The goal is to detect such non-loop allocating cases and sepa-

rate them from the rest, for which no guarantees are given. We use a fragment of a simple first-order, strict language similar to Camelot [11], with lists as only composed data-type and expressions in let-normal-form, or ANF meaning arguments to functions must be variables (k are constants, x variables, f function name):

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

We can define a non-standard type system for this language, that specifies an upper bound on heap consumption as follows ($\Sigma(f)$ is a type signature mapping function names to \mathbb{N}):

$$\frac{\vdash_H e : n \quad n \leq m}{\vdash_H e : m} \quad (\text{WEAK}) \qquad \frac{}{\vdash_H k : 0} \quad (\text{CONST}) \qquad \frac{}{\vdash_H x : 0} \quad (\text{VAR})$$

$$\frac{}{\vdash_H f(x_1, \dots, x_{n_f}) : \Sigma(f)} \quad (\text{APP}) \qquad \frac{}{\vdash_H \text{nil} : 0} \quad (\text{NIL}) \qquad \frac{}{\vdash_H \text{cons}(x_1, x_2) : 1} \quad (\text{CONS})$$

$$\frac{\vdash_H e_1 : m \quad \vdash_H e_2 : n}{\vdash_H \text{let } x = e_1 \text{ in } e_2 : m + n} \quad (\text{LET}) \qquad \frac{\vdash_H e_1 : n \quad \vdash_H e_2 : n}{\vdash_H \text{match } x \text{ with nil} \Rightarrow e_1; \text{cons}(x_1, x_2) \Rightarrow e_2 : n} \quad (\text{MATCH})$$

Finding a type of a program, based on this type system, amounts to proving that no recursive function allocates. One possibility to gain security would be to prove that the type system is sound w.r.t. a formal version of this specification. While this would not be too difficult for this simple type system, more elaborate systems are tricky to prove and the code consumer would have to trust in the correctness of the proof in order to rely solely on the types to ascertain heap space bounds.

In contrast, our approach is to develop a program logic, tuned to the case of bounded heap space, and proven sound w.r.t. a general program logic for this language. Recall that $\Gamma \triangleright e : A$ is a 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*, which captures the fact that the heap h' after the execution is at most n units larger than the heap h before execution¹: $D(n) \equiv \lambda E h h' v. |dom(h')| \leq |dom(h)| + n$. We can now write *derived rules* of the canonical form $\triangleright e : D_e(n)$ to arrive at a program logic for heap consumption:

¹We do not model garbage collection here, so the size of the heap always increases.

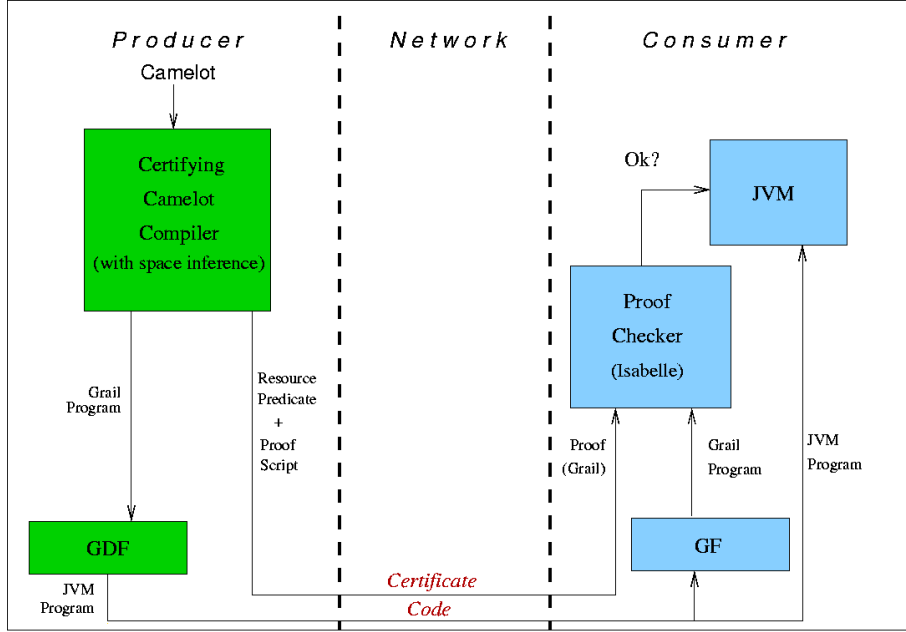


FIGURE 2. PCC infrastructure for MRG

$$\frac{\triangleright e : D(n) \quad n \leq m}{\triangleright e : D(m)} \quad (\text{DWEAK})$$

$$\overline{\triangleright k : D(0)} \quad (\text{DCONST})$$

$$\overline{\triangleright x : D(0)} \quad (\text{DVAR})$$

$$\overline{\triangleright f(x_1, \dots, x_{n_f}) : \Sigma(f)} \quad (\text{DAPP})$$

$$\overline{\triangleright \text{nil} : D(0)} \quad (\text{DNIL})$$

$$\overline{\triangleright \text{cons}(x_1, x_2) : D(1)} \quad (\text{DCONS})$$

$$\frac{\triangleright e_1 : D(m) \quad \triangleright e_2 : D(n)}{\triangleright \text{let } x = e_1 \text{ in } e_2 : D(m+n)} \quad (\text{DLET})$$

$$\frac{\triangleright e_1 : D(n) \quad \triangleright e_2 : D(n)}{\triangleright \text{match } x \text{ with nil} \Rightarrow e_1; \text{cons}(x_1, x_2) \Rightarrow e_2 : D(n)} \quad (\text{DMATCH})$$

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 side-conditions coming out of this proof will be only the inequality side-conditions 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.

3.2 Software infrastructure

The overall structure of our software infrastructure is depicted in Figure 2 and is an instance of a general PCC infrastructure [12] 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 Grail [4], an abstract fragment of Java Virtual Machine (JVM) code. 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, where 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 as high-level language Java, Scheme, or any other language that is translated to the JVM machine, and to then manually generate a proof for the desired resource property. Since the logic has been developed 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 [15] 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 Suns KVM. To tackle resource consumption of such mixed code, the foreign function calls can be annotated with their corresponding LFD types, 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 [8].

4 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.

Main novel techniques in the development of the infra-structure are our *multi-layered logics approach* for providing a logic 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.

More specifically we have produced the following

- A *completely formalised virtual machine and cost model* [5] for a JVM-like language. We have used Isabelle/HOL as theorem proving platform for this formalisation and for encoding the logics on top of it.
- A *resource aware program logic* [2] for the bytecode language of the above virtual machine.
- A *specialised logic for heap consumption* [4] that is built on top of the program logic, using our novel multi-layered logics approach.
- A *certifying compiler* for the strict, first-order functional, object-oriented language Camelot [11], integrated into a prototype proof-carrying-code infrastructure, which is available on-line [13].
- *Advanced reasoning principles* [9, 10] for resources, based on high-level type systems.

New projects that build on the MRG infrastructure are:

- EmBounded, an FET-Open STREP project funded by the EC, that aims to provide resource bounded computation for embedded systems, using Hume as high-level programming language.
- MOBIUS, an Integrated Project of the FET-GC2 proactive initiative, that deals with innovative trust management for global computing, where the resources can be network access, concurrency, and the secure flow of information.
- ReQueST, an EPSRC-funded project, that investigates the safe exchange and execution of code between nodes with large eScience databases.

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.

REFERENCES

- [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 Resources. *Theoretical Computer Science*, 2005. Special Issue on Global Computing. Submitted.
- [3] 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.
- [4] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In Andrei Voronkov Franz Baader, editor, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, volume 3452 of *LNCS*, pages 347–362, Montevideo, Uruguay, March 14–18, Feb 2005. Springer.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.

- [11] 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.
- [12] 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.
- [13] D. Sannella and M. Hofmann. Mobile Resource Guarantees. EU OpenFET Project, 2002. <http://groups.inf.ed.ac.uk/mrg/>.
- [14] 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.
- [15] 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.