A Resource-aware Program Logic for a JVM-like Language

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>

Institut für Informatik, Ludwig-Maximilans Universität, 80538 München

Abstract. Guaranteeing bounded resource consumption of mobile code is one important facet of improving the security of distributed, decentralised systems. To achieve an independent verification of resource properties we employ a proof-carrying-code approach: the mobile code is sent together with a certificate that can be checked by the consumer before executing the code. In our case, the certificate makes a statement about the resource consumption, in particular its memory consumption, and has the form of a condensed formal proof.
In this document we outline the proof-carrying code infrastructure that is being developed in the MRG project. We discuss the foundational work of developing a program logic that is powerful enough to express intensional properties of resource consumption on a JVM-like machine, yet simple enough to enable the application of automated theorem proving tools.
This logic has been encoded in the Isabelle/HOL theorem prover, and it has been proven sound and complete. The infrastructure is built on the Grail intermediate language, an abstraction over a subset of the JVM bytecode language to facilitate formalisation while retaining a close correspondence to JVM's cost model. For Grail an operational semantics is defined, and on top of that a VDM-style program logic is built that additionally tracks resource consumption such as execution time.

1 Introduction

In many distributed systems, security issues need to be addressed to provide high quality of service. For example in a Grid architecture a provider of computation resources might only make these resources available to programs that do not exceed certain limits on execution time or heap consumption. Current security mechanisms for mobile code delegate the issue of program behaviour to one of trust and cryptographic methods for signing mobile code are used for user authentication. Our approach is to directly certify properties of the program behaviour, without requiring any authentication of the code producer.

In the MRG project [19] we use proof-carrying-code (PCC) technology [13] to endow mobile code with proofs of bounded resource consumption. Thus, a service provider can easily check that a given resource policy is adhered to, and based on this rigorous proof permit execution of the code. Since such certificates are only based on the transmitted code and not on its producer, they can be independently checked, providing a scalable, decentralised security model. The feasibility of this approach relies on the observation that, while it is difficult to produce a proof of a program property, it is far less time consuming to check this property. Furthermore, in our context we are interested in resource properties, rather than more general correctness properties, which are harder to verify.

One important component of such an infrastructure is a resource-aware program logic for the language of the transmitted code. In our case we define a slight abstraction of the JVM language to decouple this level from the underlying virtual machine. We call this intermediate language Grail [4] (Guaranteed Resource Aware Intermediate Language) and it is in essence a first-order functional language. It is suitably simple to permit an encoding of a big-step, operational semantics in the Isabelle/HOL theorem prover. Because of syntactic restrictions on Grail it can also be read as an imperative language of assignments, closely modelling the costs incurred by JVM operations. The imperative reading of Grail is isomorphic to a subset of the Java Virtual Machine Language (JVML), which is why we sometimes call the logic for Grail a "bytecode" logic.

The infrastructure built in our project, requires that the logic is implemented and can be used to automatically check resource properties. Therefore we have used the theorem prover Isabelle/HOL [17] right from the start of designing the logic. This enabled us to explore various design decisions for the program logic, and we summarise the most important ones in Section 3. The use of an automated theorem prover also helped to prove soundness and completeness for the program logic. Since the program logic and its implementation is part of the trusted code base (TCB) of the PCC infrastructure, it is essential for the overall security of the system to have such results available.

2 Design of a Proof-carrying-code Infrastructure

The prototype infrastructure is shown in Figure 1. The left hand side shows the code producer and the right hand side the code consumer. 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. At the moment only the top level theorem, stating the predicted resource consumption is automatically generated. The proof itself currently has to be produced by the user within the Isabelle/HOL theorem prover. The Grail code is processed by an assembler, the Grail defunctionaliser (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.

We have used this infrastructure in certifying heap consumption of list processing programs such as sorting algorithms and are currently extending the set of test programs to other data structures such as trees. We have implemented a

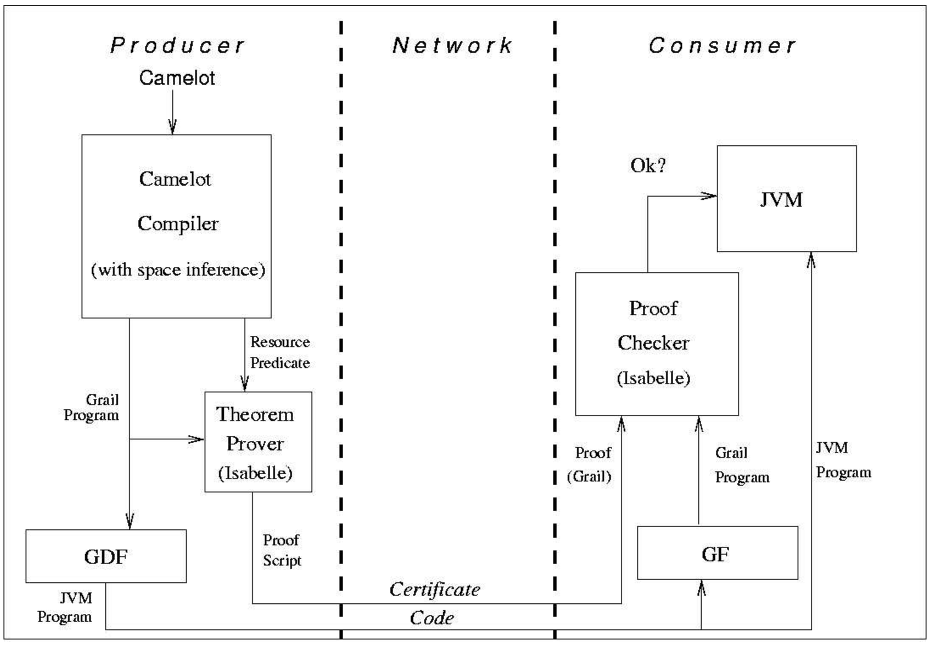

Fig. 1. PCC infrastructure with shared theorem prover and proof checker

small application to be executed on a PDA, using the MIDP standard for small devices, which provides a restricted set of Java libraries and is partially based on Suns KVM.

3 Operational Semantics and Program Logic

In this section we sketch the structure of the operational semantics of Grail and of its resource-aware program logic. We give an example of applying the logic in a list reversal function.

The operational semantics is based on the functional view of Grail, with judgements of the form

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

to be read as "in variable environment E and starting with a heap h, code e evaluates to the value v, yielding the heap h' and consuming p resources."

In developing a program logic, we consider two different styles: VDM-style [10] and Hoare-style [7]. The more commonly used Hoare-style is based on triples of the form $\{P\}\ e\ \{Q\}$ stating that if the assertion P is valid before executing the expression e, then the assertion Q is valid after execution. In order to capture intermediate values in the execution of a program, auxiliary variables are used. These variables have to be universally quantified in the formal definition of validity. In his thesis Kleymann [11] shows that a rather intricate rule of adaptation is necessary in a Hoare-style logic with auxiliary variables. A similar treatment is given by von Oheimb [20] in a program logic for a Java subset. For example the specification of the exponential function *exp*, returning its result in variable v, can be written with auxiliary variables X and Y as follows $\{0 \leq y \ \wedge\ x = X \ \wedge\ y = Y\}\ exp(x, y)\ \{v = X^Y\}$. In contrast, a VDM-style logic uses tuples of the form $e\ :\ Q$ stating that an assertion Q is valid for expression e, where Q can refer to variables in both pre- and post-state of the execution of e. Variables in the pre-state are written as "hooked" variables, e.g. $\grave{x}$, as in the specification of an exponential function $exp(x, y)\ :\{0 \leq \grave{y} \implies v = \grave{x}^{\grave{y}}\}$.

In our encoding of the program logic into Isabelle/HOL we use a shallow embedding that represents specifications as predicates in the meta-logic, rather than defining a separate data type of specifications.

A state in our language consists of a store (environment) of local variables (of type $\mathcal{E}$) and the heap (of type $\mathcal{H}$). As can be seen from the operational semantics, only the heap is modified in the evaluation of an expression, returning a value (of type $\mathcal{V}$) and a resource tuple (of type $\mathcal{R}$). The overall type of a VDM-style specification is a function returning a boolean value ($\mathcal{B}$), i.e.

$$\mathcal{A} \equiv \mathcal{E} \rightarrow \mathcal{H} \rightarrow \mathcal{H} \rightarrow \mathcal{V} \rightarrow \mathcal{R} \rightarrow \mathcal{B}$$

With this type, the informal statement "specification P is fulfilled for pre-state $(E,\ h)$, post-state $(E,\ h')$ with result value v and resource consumption p" is written formally $P\, E\, h\, h'\, v\, p\ =\ \mathsf{true}$. An expression e satisfies a specification P, written as $e : P$, iff for every (terminating) execution of e the predicate P is true, i.e.

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

The requirement that this holds for every derivable statement $e : P$ is the soundness criterion for our logic. The judgement of the program logic

$$G \rhd e : P$$

is read as "under the assumptions G, the Grail code e satisfies the specification P," where G is a set of expression-specification pairs, e is a (Grail) expression and P of type $\mathcal{A}$.

For the details of the program logic we refer the interested reader to [2].

4 Example

In this section we give an example of proving a resource property of a Grail program for insertion sort. This Grail program has been automatically produced by the compiler out of the Camelot code given below.

In our high-level language, insertion sort can be expressed as follows:

```
type iList = !Nil | Cons of int * iList
let ins a l =   match l with  Nil -> Cons(a,Nil)
                             | Cons(x,t)@_ -> if a < x then Cons(a,Cons(x,t))
                                                       else Cons(x, ins a t)
let sort l = match l with  Nil -> Nil
                          | Cons(a,t)@_ -> ins a (sort t)
```

Camelot [12] is an ML-like language and most of its syntax should be obvious. The `@_` construct is a special annotation in Camelot, which turns the `match` into a destructive `match` by returning the `Cons` cell to the freelist that is managed by the Camelot compiler explicitly. The behaviour of this freelist is also modelled in the heap inference and encoded in the program logic, which is the basis for our the automatic handling of resource consumption in our proof-carrying-code infrastructure.

The following theorem states that the heap consumption of the method `sort` in the class `InsSortM` is bounded by the length of the input list:

$$\begin{array}{l} \mathit{myTable} \ \wedge\ \forall\, l.\ \rhd\ \texttt{InvokeStatic InsSortM init l}\ :\ \mathit{initSpec}\ l \Longrightarrow \\ \forall \mathit{xs}.\ \rhd\ \texttt{LET rf l}\ =\ \texttt{InvokeStatic InsSortM init}\ \mathit{xs} \\ \qquad \texttt{IN InvokeStatic InsSortM sort [RNarg l] END} : \\ \quad \lambda\, E\, h\, h'\, v\, p\, .\ \forall\, n.\ h = \mathit{emptyheap}\ \wedge\ E = \mathit{emptyenv}\ \wedge\ n = \mathit{length}\ \mathit{xs} \\ \qquad \longrightarrow \mid \mathit{dom}\ h' \mid\ \leq\ n \end{array}$$

We assume that the method specification table contains specifications for all methods in the code. These specifications always have a specific form, which is in essence a direct translation of the meaning of the extended type system used in the inference of heap consumption in [9]. No additional knowledge about the behaviour of the program is required for proving this resource consumption. The proof for this top level theorem is then reduced to proving the specifications for each of the methods, and finally proving a theorem on the *goodContext* predicate. This predicate states that for each code-specification pair (e, P) in the method specification table, the specifications of the other methods are strong enough to prove $\rhd\, e : P$. This unusual treatment of mutual recursion is discussed in more detail in [2].

The judgement on the right hand side of the above resource theorem says that for all possible inputs, if the pre-heap h is empty, the environment E is initially empty, and the input is a list of n elements, then the size of the post-heap h' is bounded by n. Thus, the overall heap consumption is bounded by the size of the input list xs. This is exactly the upper bound on the heap consumption that we can automatically infer out of the Camelot program [9].

5 Related Work

Our program logic is most closely related to the one developed by Nipkow [16] for an imperative language and by von Oheimb [20] for a Java subset. In contrast to these logics, we use a VDM-style and handle mutual recursion directly in our derivation system, rather than developing a second derivation system over sets of code-specification pairs. Kleymann's [11] work provides the basis for many technical details in the logic and motivated us to focus on a VDM-style logic. The development of the program logic and its completeness proof are based on earlier work of one of the developers of the Grail logic [8].

Several systems use LF terms [6] as format for certificates: the Touchstone system [14, 15] with its PCC infrastructure and the ConCert system [5] for proving safety properties of an assembler language. The implicit representation of LF terms used in Touchstone to limit certificate size is similar to the compressed proof terms in Isabelle [3]. A reconstruction algorithm can retrieve a full LF term. One possibility to reduce certificate size further would be the use of Oracle strings, that guide the proof search by a non-deterministic proof checker through a logic containing structural rules. This approach has the advantage that the domain specific knowledge of the safety policy can be used to limit the certificate size: a “simple” logic will result in a highly-directed proof checker, that only rarely needs guidance by an oracle string.

Appel et al use a foundational PCC approach, that emphasises a minimal trusted code base [1]. It works directly on the operational semantics of an assembler language, proving lemmas akin to those usually defined in a program logic. Being lemmas their correctness has to be proven when stated, and no meta-theorems relating program logic with operational semantics are needed. Taking this approach no trusted VCGen is necessary. It uses a higher-order logic augmented with axioms for arithmetic, is embedded in LF and tools of the Twelf [18] system are used for proof checking.

As further reading we recommend [4] on the intermediate language Grail, [2] on the program logic, [12] on the Camelot high-level language, and [9] on the heap inference in Camelot.

6 Summary

We have discussed the foundations of an infrastructure for checking resource bounds of mobile code. Our prototype employs a proof-carrying-code approach by attaching certificates, in the form of Isabelle proof scripts, to the mobile code. These scripts can be independently checked at the consumer side before executing the mobile code. We restrict our attention to certificates of resource consumption, in particular memory consumption, so as to enable the automatic generation of these certificates out of a high level type system [9].

The logic currently used for proving heap bounds of Grail programs is simple enough to give the proof a structure of verification condition generation followed by standard simplifications, enhanced with lemmas on manipulating contexts and some knowledge about the inequalities generated by the VCGen. Since this logic is tuned for expressing heap consumption, and accurately models the memory management discipline of the high-level language, the proofs are considerably shorter than those working directly over the general program logic. By developing these more sophisticated logic in Isabelle/HOL, we can also send the soundness proof w.r.t. the underlying logic, and thus in principle obtain the small trusted code base favoured by the foundational proof-carrying-code approach.

The current prototype of this infrastructure is available as an online demonstration at: `http://lionel.tcs.ifi.lmu.de/mrg/pcc`. Details about the components of the infrastructure, the logics developed for Grail, and the high-level language can be found in the publications section of the MRG web page at: `http://groups.inf.ed.ac.uk/mrg`.

References

1. A. Appel. Foundational Proof-Carrying Code. In *LICS'01 — Symp. on Logic in Computer Science*, June 2001.
2. D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Program Logic for Resource Verification. In *TPHOL04 — Intl. Conf. on Theorem Proving in Higher Order Logics*, Park City, Utah, September 2004.
3. S. Berghofer and T. Nipkow. Proof Terms for Simply Typed Higher Order Logic. In *TPHOL'00 — Theorem Proving in Higher Order Logics*, LNCS 1869, pages 38–52. Springer, 2000.
4. L. Beringer, K. MacKenzie, and I. Stark. Grail: a Functional Form for Imperative Mobile Code. *Electronic Notes in Theoretical Computer Science*, 85(1), 2003.
5. C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *PLDI'00 — Conf. on Programming Language Design and Implementation*, pages 95–107. ACM Press, 2000.
6. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
7. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
8. M. Hofmann. Semantik und Verifikation. Lecture Notes, 1998. TU Darmstadt.
9. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symp. on Principles of Programming Languages*, New Orleans, LA, USA, January 2003. ACM Press.
10. C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
11. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, Univ. of Edinburgh, 1999.
12. K. MacKenzie and N. Wolverson. Camelot and Grail: Compiling a Resource-aware Functional Language for the Java Virtual Machine. In *TFP'03 — Symp. on Trends in Functional Programming*, Edinburgh, Scotland, September 9–11, 2003.
13. G. Necula. Proof-carrying Code. In *POPL'97 — Symp. on Principles of Programming Languages*, pages 106–116, Paris, France, Jan. 15–17, 1997. ACM Press.
14. G. Necula and P. Lee. Safe, Untrusted Agents Using Proof-Carrying Code. In *Special Issue on Mobile Agent Security*, LNCS 1419. Springer, 1998.
15. G. Necula and P. Lee. Proof Generation in the Touchstone Theorem Prover. In *CADE'00 — Conf. on Automated Deduction*, Pittsburgh, PA, June 2000.
16. T. Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In *CSL'02 — Computer Science Logic*, LNCS 2471, pages 103–119. Springer, 2002.
17. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, January 2002.
18. F. Pfenning and C. Schürmann. System Description: Twelf — a Meta-logical Framework for Deductive Systems. In *CADE'99 — Conf. on Automated Deduction*, LNAI 1632, pages 202–206, Trento, Italy, July 1999. Springer.
19. D. Sannella and M. Hofmann. Mobile Resource Guarantees. WWW page. http://groups.inf.ed.ac.uk/mrg/.
20. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.