Certification of Resource Consumption: from Types to Logic (Programming)

Alberto Momigliano and Lennart Beringer

Laboratory for Foundations of Computer Science, School of Informatics,
James Clerk Maxwell Building, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland
{amomigl1,lenb}@inf.ed.ac.uk

1 Introduction

We report on work performed in the Mobile Resource Guarantees (MRG) project [3] on developing proof-carrying code architectures for resource-related properties. The emergence of a global computing environment, where programs are routinely exchanged between principals that do not know or trust each other, has brought the issue of code verification to the forefront of attention: security-aware code consumers are simply not willing to execute downloaded code unless they can convince themselves that locally defined policies will be respected throughout the execution. Complementing mechanisms such as cryptographic techniques or code signatures that assert the authorship or the integrity of transmitted code, the proof-carrying code paradigm (PCC, [13]) has emerged as a powerful mean to guarantee code-inherent properties such as type-correct behaviour or the adherence to security policies. Contrary to earlier efforts in program verification, PCC and related technologies such as typed assembly languages (TAL, [11]) and foundational PCC [1] usually concern properties that are easier to verify than functional correctness, and that are applied on compiled rather than source code. Driven by the position of the sceptical code recipient, this paradigm requires code producers to equip programs with evidence of well-behaviour. This evidence must be transmitted as a formal certificate in a format that allows the recipient to validate its correctness by means of a fully automatic (and trusted) verification process.

The techniques used to generate certificates do not matter from the point of view of the recipient, but the ability to derive certificates during the compilation process is crucial for the success of any PCC architecture. In order to meet this goal, certifying compilers combine program annotations with sophisticated program analysis, often formulated as type systems or abstract interpretations.

Variations of the certified code paradigm have explored different formalisms in which the result of the analysis is communicated to the recipient. While the original PCC system was based on a domain-specific logic, type systems for assembly language were used in TAL. The most radical approach is foundational PCC, where the certification logic comprises only statements that can be derived from first principles: a detailed operational model that includes the machine-level encoding of instructions is formalised in a theorem prover, and certificates are only valid if their correctness w.r.t. this operational model can be derived inside the theorem prover. In effect, this approach removes domain-specific logics (or type systems in the case of TAL) from the trusted computing base (TCB), replacing it by a single component, the proof checker.

In MRG, we followed a variation of foundational PCC where additional layers mediate between the operational model and the application type system. Conceptually, the theorem prover remains a trusted component as the justification of each layer is given by its formal derivation from more basic layers. Pragmatically, however, certificates may be formulated at more abstract levels than the operational semantics, and the choice of certificate format becomes essentially a matter of negotiation between code producer and consumer. At levels where full automation is available, the performance benefits of stand-alone proof checkers may in practise justify their inclusion in the TCB, since their application can always be replaced by unfolding a certificate to a proof at a lower level of abstraction, together with an invocation of the theorem prover: at each level, a certificate may be seen as a tactic/proof script that limits the need for proof search at the lower level.

The lowest additional layer is given by a general-purpose program logic. Its role is to eliminate the repeated verification of program units through the introduction of invariants, and to serve as a platform at 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.

The second level is given by a system of *derived* assertions that represent an interpretation of a type system, possibly with respect to a particular compilation strategy. In contrast to the general-purpose logic, these derived proof systems are not expected to be logically complete, but they should provide support for automated proof search. In the case of the logic for heap consumption described below, 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. Depending on the property of interest, the second level may be further refined into a hierarchy of proof systems, for example if parts of the soundness argument of derived assertions can be achieved by different type systems.

In general, the described approach builds on the observation that type systems represent a syntactic method to determine subsets of programs satisfying particular properties, trading off logical completeness for decidability, while (program) logics are more expressive, but usually undecidable.

In the remainder of this note, we describe practical work carried out in the MRG project that instantiates the above approach to the application domain "resource consumption", and more concretely, the allocation of heap memory. Fig. 1 shows a PCC architecture built from the components developed by MRG. On the left side, the code producer generates a JVML file from a program written in our high-level programming language "Camelot", using an intermediate representation "Grail". At the same time, the type system yields a certificate that is bundled with the JVM code to obtain a certified code package. On the consumer side, the components are taken apart, the Grail code is recovered and checked against the certificate. If the check is successful, the code may be trusted and executed on a standard JVM. While this figure does not include the dependency on specific resource policies, we are currently working to-

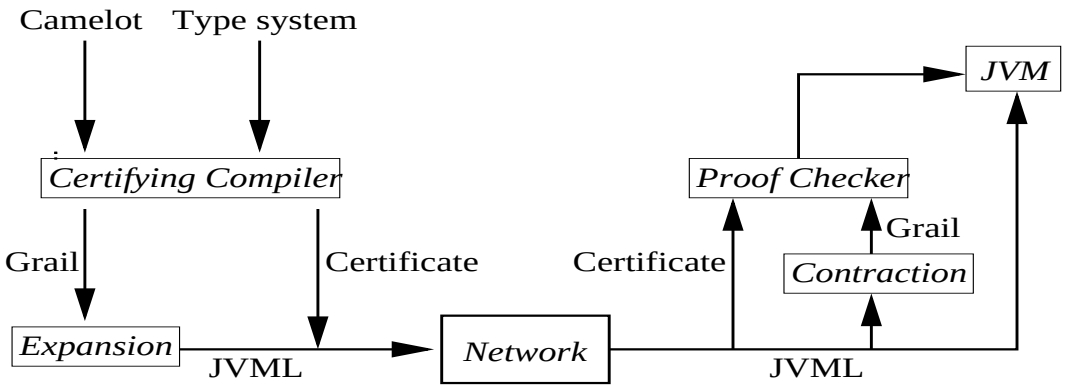

Fig. 1. MRG's PCC architecture

ward including other notions of resource consumption and admitting more flexible resource policies to be expressed and verified.

2 Components of the MRG Architecture

High-Level Language The high-level language used in MRG, Camelot [10], is syntactically, and as far as its functional semantics is concerned, a fragment of O'Caml, with recursive datatypes and functions. Differently from O'Caml, Camelot is compiled to JVM bytecode and it has been equipped with an extension that provides a smooth integration with Java methods and objects. Furthermore, it includes a memory model that is managed directly by the compiled code, rather than relying exclusively on garbage collection. All non-primitive types in a Camelot program are compiled to JVM objects of a single class `Dia` that contains fields holding nodes of any datatype mentioned in the program. Unused objects are released to a freelist, directed by language annotations such as `@_`. This primitive denotes a *destructive* match operation, i.e. the node is inserted into the freelist after the components have been extracted, so that its space can be reused during the construction of values, as is the case in the following code for insertion sort.

```
type L = Nil | Cons of int * L
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)
```

In order to ensure that objects that are inserted into the freelist cannot be aliased, various type systems may be imposed on Camelot programs. In the remainder of this paper, we assume a linear typing regime, although more permissive disciplines may be considered as well [4].

Inference of Heap Space Consumption In order to analyze the memory requirements of functional programs, Hofmann and Jost [7] introduced a type system that solves the following problem. Given a program *e* of type, say, `L -> L`, calculate a (linear) function f such that computing *e(l)* for some input list *l* will not require more than *f(|l|)* additional heap cells, provided that a freelist is available for storing temporarily unused cells. In the context of Camelot compilation, the result of such an analysis can be used to ensure that the evaluation of *e(l)* will not perform object creation (JVM instruction new), by wrapping the evaluation with code that allocates a freelist of the sufficient length *f(|l|)* prior to calling *e*.

The analysis of [7] is formulated using an extended notion of types such that different portions of the input can contribute a different amount to memory consumption. For each datatype constructor, a numeric annotation indicates the amount of heap that is required for a single build operation using that constructor. Judgments are of the form $G,n \vdash e : T,m$ where T is the (extended) type of e with respect to the context G, while n and m represent constants describing memory requirements. Numeric annotations in T are interpreted relative to the size of the *output* data structure, and the annotations in G refer to the size of the *input* data structures. Some of the typing rules are given in Fig. 2. The effect of a pattern match on the freelist depends on whether the match is performed destructively or not. In

$$\frac{n \geq 1+k+m}{G,h:\mathsf{int},t:\mathsf{L}(k),n \vdash \mathsf{Cons}(h,t):\mathsf{L}(k),m}\ \textsc{Cons}$$

$$\frac{G,n \vdash e_1 : A,m \qquad G,h:\mathsf{int},t:\mathsf{L}(k),n+1+k \vdash e_2 : A,m}{G,x:\mathsf{L}(k),n \vdash \mathsf{match}\ x\ \mathsf{with}\ \ \mathsf{Nil} \Rightarrow e_1 |\ \mathsf{Cons}(h,t)@\_ \Rightarrow e_2 : A,m}\ \textsc{DMatch}$$

$$\frac{G,n \vdash e_1 : A,m \qquad G,h:\mathsf{int},t:\mathsf{L}(k),n+k \vdash e_2 : A,m}{G,x:\mathsf{L}(k),n \vdash \mathsf{match}\ x\ \mathsf{with}\ \ \mathsf{Nil} \Rightarrow e_1 |\ \mathsf{Cons}(h,t) \Rightarrow e_2 : A,m}\ \textsc{Match}$$

Fig. 2. Selected rules from [7]'s typing system

both cases, the branch executed in the case of an empty list has exactly the same memory behaviour as the composite expression. In the case of a non-empty list, the freelist available for the continuation grows at least by the amount k "stored in" the list node that is taken apart. In case of a destructive match, the cell inhabited by this node becomes available as well, which explains the additional increment in rule DMATCH.

The inference process, called LFD, for the type system consists of two stages. First, a skeleton type derivation is constructed where the numerical annotations *n, k, m,...* are interpreted as (rational) variables, constrained by side conditions such as the one in rule CONS. These side conditions are collected, and in a second step handed over to a linear programming solver. Any feasible solution to the linear program corresponds to a possible typing derivation.

In the context of certificate generation, the solution inferred by the analysis (if existing) is presented as a signature that contains one typing for each Camelot function. In the case of our example program, one such signature is

$$\{\mathsf{ins} : 1, \mathsf{int} \times \mathsf{L}(0) \to \mathsf{L}(0), 0,\ \mathsf{sort} : 0, \mathsf{L}(0) \to \mathsf{L}(0), 0\}$$

For both functions, this signature asserts that the heap consumption does not depend on the size of the input: ins consumes one heap cell, while sort executes in-place: the cell that is required in the call to ins has previously been gained in the pattern match.

Abstract Representation of Bytecode As the target of the Camelot compilation we introduced a representation of bytecode, Grail [6], that abstracts from details of virtual machines of little significance for our purpose, such as the operand stack. Mediating between the high-level language and concrete bytecode formats such as JVML or MSIL, Grail is equipped with a functional as well as with an imperative operational semantics. The former is used during the cost-transparent compilation, while the latter admits a reversible expansion into concrete bytecode. Thus, we can apply existing code transmission formats and execution machinery without being tied to one particular code format. Certificates of resource consumption are formulated at the Grail level, based on a cost model the operational semantics can be parametrized by. The formalisation is based on the functional interpretation of Grail expressions e and is built from judgements of the form

$$E \vdash h, e \Downarrow h', v, r$$

where E is an environment mapping program variables to values, h is the initial heap, h' the final heap, v the result value, and r the cost incurred during the execution.

The compiler translates a program into Grail following a whole-program compilation approach. The resulting code contains one (static) method for each Camelot function. For example, the Grail code for insertion sort includes the method InsSort.ins, with the following (slightly pretty-printed) body:

```
method InsSort.ins(int a, Dia l) = call f
 f : if isNull(l) then l = null ; Dia.make(a, l)
                  else v3 = l.HD ; v2 = l.TL ; Dia.free(l) ;
                       if a < v3 then l = Dia.make(v3, v2) ; Dia.make(a, l)
                                 else l = InsSort.ins(a, v2) ; Dia.make(v3, l)
```

Destructive datatype match operations lead to the insertion of the cell into a freelist (method free), and construction of a value draws a cell from the freelist (method make). The implementation of make is such that a new object is only allocated if the freelist is empty - in fact, this is the only place at which object creation occurs in the compiled code. Consequently, we may establish in-space execution if we can verify that make will never be called on an empty freelist. The verification of this property is based on a general-purpose program logic that we describe next.

Program Logic Although previous program logics have been developed for Java-like languages [8, 9], we found the abstractions introduced by Grail, and the type structure available via the Camelot compilation, could best be exploited in a program logic that is specific to Grail and resource certification. Aiming to separate the verification of general resources from the issue of termination, we designed a logic of partial correctness, where the traditionally separated pre- and post-conditions are combined into single assertions. Judgments are of the form

$$G \rhd e : P$$

where e is a Grail expression, P a predicate over the semantic components (environment, initial heap, final heap, result value, cost component) of the operational semantics, and G a context that collects specifications for recursive methods. Thus, relations between the initial and the final content of heap cells can be expressed inside the definition of P, without the need to introduce auxiliary variables, and our rule for assignment avoids the (at first sight) counterintuitive modification in the pre-condition characteristic to previous program logics. As an example rule, we give the rule for sequential program composition

$$\frac{G \rhd e_1 : P_1 \qquad G \rhd e_2 : P_2}{G \rhd e_1 \,;\, e_2 : (\lambda\, E\ h\ h'\ v\ r.\ \exists\, r_1\ r_2\ h_1\ .\ (P_1\ E\ h\ h_1\ \bot\ r_1)\ \wedge\ (P_2\ E\ h_1\ h'\ v\ r_2)\ \wedge\ r = r_1 @ r_2)}\ \text{VCOMP}$$

where @ is a binary combinator on resources. As derived rules, we obtained cut elimination rules that allow us to prove program logic judgements that do not depend on contextual specifications, and rules that allows the arguments of method calls to be adapted when extracting an assumption from the context.

Traditionally, one could argue that soundness and completeness of program logics are mostly of meta-theoretic interest. In a proof-carrying code architecture, they are of far greater importance. This obviously applies to soundness, as the code consumer bases the decision whether to accept a transmitted program on the validation of the certificate. Completeness, on the other hand, ensures that the logic is powerful enough to support not only our current high-level type system, but any type system that has a semantic interpretation depending only on the five semantic components E, h, h', v and r. In order to achieve this confidence in our logic, we formalized the operational semantics and the program logic in the theorem prover Isabelle [16] and performed the proofs of soundness and completeness relative to the ambient logic HOL [2].

Certificate Generation and Verification As motivated in the introduction, certificate generation can be obtained by giving semantic interpretations of type systems in the program logic, and proving analogs of the typing rules as derived lemmas. In the case of the LFD type system discussed above, this results in assertions of a fixed syntactic form

$$[\![m, D \blacktriangleright T, n]\!]$$

where D is a linear typing environment assigning numerically annotated types to variables in e, T is a annotated type, and m,n are numbers. Such assertions expand into an ordinary assertion in the program logic, which expresses the semantic meaning of the typing judgement in the high-level system as described above. Building upon the formalisation

$$\frac{G \rhd e_1 : [\![n, D_1 \blacktriangleright \mathbf{1}, m]\!] \quad G \rhd e_2 : [\![m, D_2 \blacktriangleright T, k]\!]}{G \rhd e_1 \,;\, e_2 : [\![n, (D_1, D_2) \blacktriangleright T, k]\!]}\ \text{DAComp} \qquad \frac{D(y) = \mathbf{L}(k) \quad D(x) = \mathbf{I}}{G \rhd \mathsf{Dia.make}(x,y) : [\![m+k+1, D \blacktriangleright \mathbf{L}(k), m]\!]}\ \text{DAMake}$$

$$\frac{G \cup \{(\mathsf{call}\, f, P)\} \rhd \mathit{funtable}\, f : P}{G \rhd \mathsf{call}\, f : P}\ \text{DACall}$$

$$\frac{G \rhd e : [\![n+k, (D, h : \mathbf{I}, t : \mathbf{L}(k)) \blacktriangleright T, m]\!]}{G \rhd h = x.\mathsf{HD} \,;\, t = x.\mathsf{TL} \,;\, e : [\![n, (D, x : \mathbf{L}(k)) \blacktriangleright T, m]\!]}\ \text{DAMatch}$$

$$\frac{x \neq t \qquad G \rhd e : [\![n+k+1, (D, h : \mathbf{I}, t : \mathbf{L}(k)) \blacktriangleright T, m]\!]}{G \rhd h = x.\mathsf{HD} \,;\, t = x.\mathsf{TL} \,;\, \mathsf{Dia.free}(x) \,;\, e : [\![n, (D, x : \mathbf{L}(k)) \blacktriangleright T, m]\!]}\ \text{DADMatch}$$

Fig. 3. Sample derived assertions

of the general-purpose program logic, we formally *derived* such rules in Isabelle/HOL, including rules for the freelist management methods and destructive and non-destructive match operations. Some of them are depicted in Fig. 3.

Depending on the implementation of the certificate checker, various certificate formats can be considered. In fact, there are several aspects where logic programming techniques are playing a role in PCC: to begin with, safety properties have been expressed as deductive systems, which, as popularized by systems such as Lambda Prolog [12] and Twelf [17], can be seen as logic programs. Further, and somewhat ironically, considering the reputation that declarative programming has in certain circles, goal-oriented proof search is now being exploited to make PCC more *efficient*. Indeed, in the original PCC approach, certificates were full LF proof terms, so as to make the proof-checker at the consumer side as simple as possible [13]. As this yields certificates that are several orders of magnitude larger than their accompanying programs, more efficient representation of such terms were devised, where (higher-order) unification was used to reconstruct the necessary information at the client side [14]. This represented a slight shift of burden from the producer side, and it was refined once it was realized that proof checking at the consumer side could be accomplished with a very simple and trusted (non-deterministic) logic programming interpreter. In both the foundational [18] and the oracle based approach [15], the certificate consists of a goal (the safety predicate) to be executed against the logic program embodying the safety policy. A hint (such as loop annotations) or an oracle bit-stream allows the consumer to replay the proof *without* any backtracking.

In the MRG architecture, similarly to the foundational one, the heap logic does not need to be trusted, but its soundness proof can be verified once and for all by the consumer (most likely in the guise of *off-device* verification). Once installed, the syntax-oriented rules of the heap logic may be used as a logic program. In the current implementation the soundness proof is directly exploited and we designed a hand-tailored Isabelle tactic [5] which performs goal-oriented proof-search on the derived rules, while at the same time checking that any arising arithmetic and set-theoretic constraint is met. Due to both the Isabelle inductive setup (requiring judgments such as provability in the program logic to be monotonic) and the need to stay close to the JVM model, the encoding is first-order and the verification at the moment far from efficient. However, similar to the emergence of Prolog from general resolution, there is much to be gained from switching from tactical theorem proving to logic programming, not only in pragmatic terms, but in a dramatic reduction of the trusted code base. Current research is focused on providing a Twelf implementation of the heap logic, using hypothetical and parametric judgments to encode explicit contexts such as G and D, much as in [18]. We further aim to generalise our approach to an architecture with more complex resource policies that involve high-level type systems for properties different from heap consumption.

Acknowledgements: This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET pro-active initiative on Global Computing. The paper is based on joint work with David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, Ian Stark, Kenneth MacKenzie, Nicholas Wolverson, Matthew Prowse, Hans-Wolfgang Loidl, Olha Shkaravska and Steffen Jost.

References

1. A. W. Appel. Foundational proof-carrying code. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE Computer Society Press, June 2001.
2. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proceedings of the 17th International Conference on Theorem Proving in Higher-Order Logics, (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 34–49. Springer, 2004.
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: Proceedings of the International Workshop CASSIS 2004*, number 3362 in Lecture Notes in Computer Science, pages 1–26. Springer-Verlag, 2005.
4. D. Aspinall and M. Hofmann. Another type system for in-place update. In D. Le Métayer, editor, *Programming Languages and Systems (Proceedings of ESOP 2002)*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.
5. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In A. Voronkov & F. Baader, editor, *Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005. Proceedings*, volume 3452 of *Lecture Notes in Computer Science*, pages 347–362. Publisher: Springer-Verlag, Feb 2005.
6. L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. In V. Sassone, editor, *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, number 85.1 in Electronic Notes in Theoretical Computer Science. Elsevier, June 2003.
7. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, volume 38 of *ACM SIGPLAN Notices*, pages 185–197, New York, January 2003. ACM Press.
8. M. Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
9. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.* To appear.
10. 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.
11. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.
12. G. Nadathur and D. Miller. Higher-order logic programming. In C. H. D. Gabbay and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.
13. G. Necula. Proof-carrying code. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 5, pages 177–220. MIT Press, 2005.
14. G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.
15. G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL01), London*, January 2001.
16. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Jan. 2002.
17. F. Pfenning and C. Schürmann. *Twelf User's Guide*, 1.4 edition, Sept. 2004. Available at `http://www.cs.cmu.edu/~twelf/guide`.
18. D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 264–274, New York, NY, USA, 2003. ACM Press.