2009 Fourth Latin-American Symposium on Dependable Computing

A Proof-carrying-code Infrastructure for Resources

Hans-Wolfgang Loidl*, Kenneth MacKenzie†, Steffen Jost‡ and Lennart Beringer*
* Institut für Informatik, Ludwig-Maximilians Universität, D-80538 München, Germany;
Email: {hwloidl,beringer}@tcs.ifi.lmu.de
† Laboratory for the Foundations of Computer Science,
School of Informatics, The University of Edinburgh, Edinburgh EH8 9AB, Scotland;
Email: kwxm@inf.ed.ac.uk
‡ School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SX;
Email: jost@cs.st-andrews.ac.uk

***Abstract*—This paper tackles the issue of increasing dependability of distributed systems in the presence of mobile code. To this end we present a complete Proof-carrying-code (PCC) infrastructure for independent and automatic certification of resource bounds of mobile JVM programs. This includes a *certifying compiler* for a high-level language, which produces a certificate of bounded heap consumption, and independent *certificate validation*, realised via proof-checking, on the code-consumer side. Thus, we are now in a position to automatically infer linear upper bounds on the heap consumption of a strict, first-order functional language, generate a certificate encoding a formal proof of such bounded heap consumption and independently validate this certificate at the consumer side by checking the certificate. This prevents mobile code from exhausting resources on the local machine.**

***Keywords*-proof-carrying-code; resource analysis; program verification;**

I. Introduction

Ensuring dependability of distributed systems in the presence of mobile code is a significant challenge in the design of complex systems. In particular, the design should ensure that mobile code, for example downloaded from the internet, does not harm the local machine. Most approaches to tackle this challenge build on an authentication mechanism, which ensures the identity of the alleged author of the code. However, in the end the decision whether or not to execute the code is delegated to the user: only if he/she trusts the author in not providing (inadvertently) malicious code, the execution of the program will be permitted.

Our approach to the design of dependable, distributed systems does not rely on such a trust relationship. Instead the mobile code is directly checked for possibly harmful behaviour, and execution is only permitted if the code comes with a certificate of not being malicious. To achieve this, we use a proof-carrying-code (PCC) approach [1], where mobile code is transmitted together with a certificate. In particular, we are interested in the bounded resource consumption of mobile code, and we present a PCC infrastructure for resource bounded code. On a foundational level this requires a program logic to formalise the properties of interest. On a system level this requires tools for automatically generating a certificate and for checking the certificate attached to a piece of code. In this paper we present a complete infrastructure for proof-carrying-code on distributed systems based on the Java Virtual Machine (JVM).

The formal basis for proving resource bounds on JVM code consists of a resource-aware program logic for (a subset of) the JVM machine that has been proven sound and complete w.r.t. an operational semantics for the JVM. On top of this logic we have developed a heap space logic, which is tailored for proving statements on heap consumption. This logic is used as the basis for automatic certificate generation. In the validation phase we use our formalisation of these logics in Isabelle/HOL together with a generic tactic for checking the certificate.

Our main contribution is the design and implementation of an entire PCC infrastructure for the safe execution of resource bounded code. Examples demonstrate the feasibility of this foundations-driven approach and the advantages compared to authentication based approaches. The main, novel components in the implementation of the infrastructure are:

- The *automatic inference of linear heap bounds* for the functional language Camelot (this builds on the standalone inference presented in [2]).
- The *automatic generation of certificates* by the compiler.
- The *validation of a certificate*, which is a condensed version of a formal proof.

The infrastructure presented in this paper uses commonly available tools and OpenSource software to make the results of our research widely available. It is available as an on-line demo, together with a user's guide and a set of exercises.

II. Related Work

A. Proof-carrying-code.

The concept of proof-carrying-code (PCC) originated in the doctoral thesis of George Necula [3], where he developed the logical theory of PCC and then applied it in the development of a certifying compiler for a subset of C. This work was extended and generalised in a series of papers: in [4], Necula and Lee use PCC to certify the memory safety of operating system kernel extensions written in assembly language; in [1] (the standard PCC reference), Necula describes the use of PCC to implement safe assembly-language extensions to a runtime system for Standard ML; in [5], a compiler from

978-0-7695-3760-3/09 $25.00 © 2009 IEEE
DOI 10.1109/LADC.2009.13

Java to annotated x86 assembler language is described, which adds information on loop invariants such as register types, variable bounds, and modified registers, to prove that the code is well-behaved. Technical details of the logical methods used in Necula's PCC implementation can be found in [6]. Necula and Lee also consider mobile programs in [7], where PCC is applied to the problem of checking the security of untrusted mobile agents scanning a database.

Necula's approach (sometimes referred to as *Classical PCC*) is based on having a specialised logic for the property one is interested in, together with programs called a *proof checker* and a *verification condition generator* (VCG) for the logic (see [1] for details). These components are tailored to the logic, and must be reimplemented when a new logic is introduced. Moreover, they form part of the *Trusted Code Base* (TCB): the soundness of the PCC system depends on the correctness of these components, and errors in implementation could compromise the system and allow unsafe programs to be executed. Similarly, one must also assume that the logic does in fact correctly describe the required safety property. To deal with these issues, Appel [8] has proposed the concept of *Foundational PCC*, in which the operational semantics of the target machine are encoded in a theorem prover, and safety proofs are expressed with respect to this formalisation. This approach reduces the TCB to a minimum, but increases the size of proofs, which may lead to scalability problems. Applications and further developments of FPCC can be found in [9], [10], and also in several publications on the FLINT project, e.g. [11]–[13].

A different approach to the logical foundations of PCC has been explored by Wildmoser and Nipkow [14], who develop a PCC framework involving a VCG which has been formally verified in the Isabelle/HOL theorem prover. This framework is parametric and can be instantiated with respect to a specific programming language and safety property: for example, in [15] it is applied to safety properties for arithmetical operations in Java bytecode.

An interesting variation on the PCC idea is *Abstraction Carrying Code* [16], in which mobile programs are equipped with fixpoints of static analyses which then enable the code consumer to check a safety property simply by checking that the annotations are in fact fixed under appropriate operations.

B. Resource Analysis

The problem of statically predicting resource usage has a long history, with early work mostly focusing on prediction of execution time. In 1975 Wegbreit [17] developed a system to automatically find bounds for execution times of LISP programs, based on systems of recurrence equations. Further work on LISP was carried out by Le Métayer [18] and Rosendahl [19]. Cost inference for logic programs has been studied in [20] and elsewhere. A great deal of work has also been done on worst case execution time (WCET) for real-time systems: see the recent paper of Wilhelm et al. [21] for a comprehensive survey.

Research on memory allocation is more recent, and initially focused on functional languages: see for instance the sized types of Pareto and Hughes [22], or the languages of Hofmann [23] and Aspinall-Hofmann [24], where type systems enforce good behaviour for heap allocated data structures. The type system of Hofmann and Jost (described below) allows static prediction of memory bounds, and has recently been extended by Campbell [25] to the context of stack usage. Another functional language where various resource-prediction schemes have been developed is the Hume language, which will be discussed later. More recently, there has been work on memory prediction for object-oriented languages such as Java. One approach is due to Hofmann and Jost [26], and involves the concept of *amortised* costs; the AHA project [27] at Nijmegen proposes to extend this work to Java and to produce an implementation. An interesting recent development is the use of geometric methods to predict Java memory usage by Braberman et al. [28], [29].

III. THE PCC INFRASTRUCTURE

We start with a short characterisation of the languages involved in the infrastructure:

- *Camelot* is the high-level language, used by the programmer to generate resource-safe code [30]. It is a strict, first-order language, similar in syntax to ML and with object-oriented extensions. These extensions allow the use of a destructive match operator, with an implicit free-list management of the reclaimed cells. Additionally, reclaimed cells can be named and re-used, thus enabling to code in-place operations, e.g. for sorting, in Camelot.
- *Grail* is a subset of the JVM language [31], covering a large fragment of the sequential instructions that exist in the JVM. Because it contains more structural information about the programs, for example tail-recursive functions encoding goto-based loops in the JVM, it is more suitable for the development of a program logic, which is a central piece of our infrastructure. Since it covers almost all JVM instructions, it can also be used on most plain JVM code, i.e. code not generated through compilation from Camelot. The latter opens the possibility for manually, or semi-automatically, generating certificates for such code.
- *JVM bytecode* is used for transferring mobile code. There is a one-to-one correspondence between Grail and our JVM fragment, and thus Grail code can be reconstructed out of the JVM class file on the consumer side.
- *Isabelle proof scripts* are used as the encoding of the certificate generated by the certifying compiler. We will show in Section IV that the information required in the proof script is small, mainly invariants for the functions.

The prototype infrastructure is shown in Figure 1. The left hand side shows the code producer with a certifying compiler as its main component. The right hand side shows the code consumer with a proof checker as its main component.

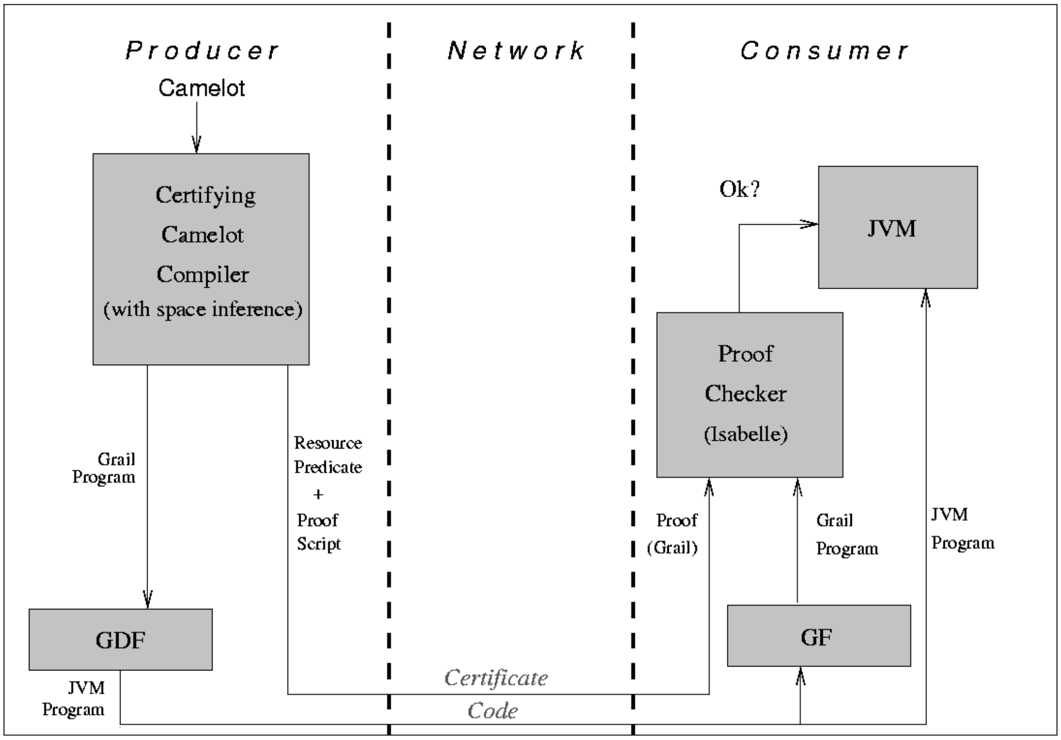

Fig. 1. A complete proof-carrying-code infrastructure for resources

A. Certifying Compiler.

On the producer side a *certifying compiler* translates high-level Camelot programs into the Grail intermediate code and additionally generates a certificate of its heap consumption. To construct the latter, a type-system based space inference is performed, which infers linear upper bounds on the heap consumption of the Camelot program (see Section III-B). The result of this inference is translated into a statement on bounded resource consumption, formalised as a lemma in the heap space logic for Grail [32]. A close correspondence between this analysis and the heap space logic, that is used for encoding the generated certificates, is crucial for the simplicity of the certificates. In essence the certificate validation phase can replay the same proof rules on the logic level, that have been used in the heap space inference on the Camelot level. 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 consumer.

B. Automatic inference of heap consumption.

The inference on heap consumption is an implementation of the static program analysis of Hofmann and Jost [2]. The method allows the fully automatic determination of upper bounds on heap space consumption, where the bound may linearly depend on the input sizes of a program.

This type-based analysis is performed in two steps: in the first step a standard type derivation is augmented with constraints in the form of inequalities over rational numbers; in the second step these constraints are solved using a standard linear programming package (the `lp_solve` package [33]). This approach leads to a very efficient analysis that scales well for increasingly complex programs.

The key to any feasible program analysis lies in trading precision for clarity and efficiency in a sensible manner, since it is generally infeasible to track all possible program states. The solution employed here is especially radical, since we abstract the entire machine state into a single, non-negative rational number, referred to as the *potential* of the machine state.

It is important to note that one will never actually compute this number, the potential, for any actual machine state other than the initial state. Instead, the analysis only tracks the relative effect of a program step on the overall potential. The idea is to assign the potential in a clever way such that for each instruction in each possible machine state, the change in potential caused amortises the actual cost for that instruction at that particular machine state. Thereby, the *amortised cost* of an instruction, i.e. the actual cost plus the difference in potential caused, can become a constant independent of the machine state, if the potential is cleverly chosen. The cost of executing a program is then simply the sum of the (constant) amortised costs plus the potential of the initial state, which is shown in the examples in Section IV.

The manual application of this technique is known in complexity theory as the “Amortised Analysis” technique [34].

The significant challenge of applying it lies in finding the abstraction of the machine state into the potential. This problem is solved in our system by using linear programming, at the expense of restricting the potential so that it must depend linearly on the sizes of the input data (a restriction which is not inherent to the amortised analysis). The benefit of linear constraints is of course that highly efficient solvers are readily available.

An important factor in the embedding of this inference into the infrastructure is that the potential models the memory free-list as used in the Camelot compiler (see [30]). The potential represents an upper bound on the free-list size, expressed in terms of the existing data objects. In a coarse sense, each data object is associated by the static analysis with a certain portion of the free-list. Processing a particular data object may only use its associated portion of the free-list, although there is no requirement to track these notions at runtime.

C. Logics

The formal basis for our infrastructure is a hierarchy of logics (discussed in more detail in [35]).

The *heap space logic* is a domain-specific derivation system, tailored to reason about heap consumption. Its rules amount to Grail-level interpretations of the high-level typing rules in Camelot. Judgements take the form $\Gamma \rhd e : [\![U, n, [G] \blacktriangleright T, n']\!]$, relating a (Grail) expression e in some proof context Γ to its type T (an interpretation of an extended Camelot-type), initial and final constant heap size descriptions n and n', and a Grail-level (linear) interpretation (U, G) of a Camelot typing context, mapping variables to extended types. The proof rules over such *resource assertions* have only simple side-conditions and therefore simplify proofs performed on this level.

The general purpose *program logic* can be used to reason about functional correctness as well as resource consumption. It has judgements of the form $\Gamma \rhd e : A$. Assertions A in this logic are predicates on the components of Grail's operational semantics, namely a variable environment, a pre-heap, a post-heap, and a result value.

The resource-aware *operational semantics* is the basis of our hierarchy and formally models the execution of Grail code together with its resource consumption. Both a functional and an imperative semantics, as well as a cost model, have been developed. We have proven in previous work the correspondence between the functional and an imperative semantics [31].

Crucial for the safety of our infrastructure is the encoding of all levels of this hierarchy into the Isabelle/HOL theorem prover. Based on these formalisations we have proven soundness and completeness of the program logic w.r.t. the operational semantics, using the Kleymann-Nipkow-Hofmann technique [35]. We have also proven soundness of the heap space logic w.r.t. the program logic. Because of these general soundness results, the operational semantics is the only component of the hierarchy that is part of the trusted code base. If the consumer does not trust the heap space logic, it would be possible, in principle, to send the soundness proofs together with the certificate and check these soundness statements, too. Thus, our formally-grounded design helps to minimise the trusted code base in a similar style as in the foundational PCC approach [8].

Certificates are represented as Isabelle/HOL proof scripts, operating on the highest, namely the resource logic level. They encode a condensed summary of the analysis result, additional control flow information, and formal, automatically generated lemmas. These proof scripts make use of pre-implemented verification tactics, and thus reduce the size of the certificate considerably. Since a certificate may (on request) be expanded to a more foundational guarantee in any of the lower level formalisms, we combine the advantages of classic PCC (ease of reasoning) with those of foundational PCC (formalised soundness of the logic and a small trusted code base). Another benefit of the higher certificate levels concerns proof complexity. Each justification of a formalism with respect to its predecessor hides verification complexity, for example by instantiating existential quantifiers once and for all, or by prescribing a particular (in our case: linear) interpretation of Camelot contexts.

Although the implemented infrastructure restricts the input language to one with lists and trees as composed data types and only deals with heap space consumption, we have in the meantime extended the formalisation of the logic to cover arbitrary data types, and to deal with resources in a more abstract way, encoding for example time consumption and traces of function calls.

D. Validation.

On the consumer side, the Grail code is retrieved via a disassembler, the Grail functionaliser (gf). The proof checker is Isabelle/HOL, used in batch mode, to automatically validate the certificate w.r.t. a resource bound that is specified by the code consumer. An overall resource statement establishes that the proven bound is smaller than the specified bound. Once this has been confirmed the code can be executed safely on the consumer side.

One basic design decision for our PCC infrastructure was to minimise the size of the generated certificates and to accept a comparatively large trusted code base in return. This is based on our observation that previously reported certificate sizes pose a considerable problem in the acceptance of PCC in general. We believe that a large trusted code-base with a fairly heavy-weight certificate validation phase is acceptable in large-scale, distributed environments such as Grid architectures [36], where individual machines are very powerful already.

IV. Worked Example

The following small example shows key aspects in the process of analysing and compiling a Camelot program, transferring it and validating the resource consumption on the consumer side. The example is an in-place insertion sort program over a user-defined list structure. The annotation in the `!Nil` constructor indicates that this value is represented as a nil-pointer rather than a pointer to a heap location.

```
(* define lists of integers *)
type iList = !Nil | Cons of int * iList

(* insert a into the list l *)
let ins a l = match l with
             Nil -> Cons(a,Nil)
           | Cons(x,t)@_ -> (* NB: destructive match *)
                            if a < x
                              then Cons(a,Cons(x,t))
                              else Cons(x, ins a t)
(* insertion sort (in-place) *)
let sort l = match l with
            Nil -> Nil
          | Cons(a,t)@_ -> (* NB: destructive match *)
                           ins a (sort t)
```

The matches annotated with @_ represent destructive match operations: the heap cell that was matched against is returned to the free-list before entering the right-hand side of the Cons branch. Thus, the application of Cons on the right hand side can re-use this cell. In the case of ins, this means that only one additional cell is required in the then branch, and that no additional space is consumed in the else branch (with any additional cells available being handed down to the recursive call). Since the sort function also uses a destructive match and makes one cell available when calling ins, the overall heap consumption of sort is 0.

A. Space Inference:

The analysis, as discussed in Section III-B, infers an annotated type, representing a linear closed cost formula bounding a program's memory usage. For example, if the type of a function processing a list of integers is specified as $(x, \mathsf{List}[\mathsf{int}, \#, y] \rightarrow \mathsf{List}[\mathsf{int}, \#, u], v)$ then executing that function requires *at most* $x + y \cdot |l|$ memory cells to compute, where $|l|$ refers to the length of the input list.

Note that list types are *not* built into the analysis. Hence the name of a datatype is followed by a list enclosed in square brackets and separated by |, showing all argument types for each constructor. The symbol # recursively stands for the enclosing type itself, with *identical* annotation values. Each constructor carries its own annotation. For another example a type like $\mathsf{Bool}[17|4]$ among the argument types tells us that executing this function with False requires up to 17 memory cells, but executing it with True requires only up to 4 cells.

Furthermore, we can also see that the list processing function shown above returns *at least* $u \cdot |l'| + v$ heap cells after the computation, where $|l'|$ denotes the length of the *result*. These memory cells may stem from destructive matches on input or intermediate data, or they are handed down unused from the memory cells initially requested.

```
Resource constraints: 26 inequalities in 25 variables.

Solution yields the following enriched types:
ins : 1, int -> List[0|int,#,0] -> List[0|int,#,0], 0;
sort: 0, List[0|int,#,0] -> List[0|int,#,0], 0;

Total processing time: 0.01 seconds.
```

Shown above is the output of the analysis for the insertion sort example. Most annotations are zero, so sort does not require any additional memory resources during computation. In other words, the list is sorted in place. We also see that calling ins requires a memory cell regardless of the actual input. However, this information is only relevant if we would call ins directly, since the type of sort already accounts for all subsequent calls. The memory cell requested by ins is gained from the initial destructive match performed by sort, as we have seen in the initial discussion of this example.

B. Certificate Generation:

In the next step the certifying compiler emits Grail code, which is then translated down to JVM code, and also a certificate. The latter is a fragment of an Isabelle proof script, but contains only those pieces of information that cannot automatically be reconstructed on the consumer side. As a consequence we have *light-weight* certificates, which are proven by a heavy-weight proof checking environment. In our example the main part of the *certificate* looks like this:

Dominates		
InsSort · ins ↦ false	InsSort · sort ↦ false	f_1 ↦ false
Spec		
InsSort · ins	↦	$[\![\{l\}, 1, [\{l \mapsto \mathbf{L}(0)\}] \blacktriangleright \mathbf{L}(0), 0]\!]$
InsSort · sort	↦	$[\![\{l\}, 0, [\{l \mapsto \mathbf{L}(0)\}] \blacktriangleright \mathbf{L}(0), 0]\!]$
Context		
InsSort · ins($[a,\ r2]$)	↦	sMST InsSort · ins $[a,\ r2]$
InsSort · sort($[r1]$)	↦	sMST InsSort · sort $[r1]$
InsSort · ins($[v2,\ l]$)	↦	sMST InsSort · ins $[v2,\ l]$

The first table, Dominates, carries information about the control flow of the program for user-defined and automatically generated functions. It is used to improve the performance of the certificate check, avoiding a duplication of proofs that might be needed otherwise. Since our compiler is a whole-program compiler, it generates only one class (InsSort) for the user-defined code and all methods inhabit this class. The second table, Spec, defines a table mapping JVM methods (represented as class, method pairs) to their resource assertions, carrying information about space consumption. These resource assertions are directly generated from the annotated types produced by our space inference. Its components (for InsSort · ins) encode the set of free variables ($\{l\}$), the constant component of the heap space bound (1), a typing context mapping all free variables to annotated types ($\{l \mapsto \mathbf{L}(0)\}$), the annotated result type of the function ($\mathbf{L}(0)$), and the constant component of the free heap space bound after execution (0). The third table, Context, maps all call sites in the program to their corresponding resource assertions. To obtain the latter, the sMST function essentially performs a lookup on the Spec table. Additionally, the environment in the assertion is initialised with bindings for the specified arguments. The structure of this table is fairly schematic and could also be generated automatically. The essence of the certificate are the resource assertions in the Spec table.

In total, 3 files are transmitted in a JAR file from the producer to the consumer:

- An application class file, with the JVM code of the Camelot program
- A memory management class file, with the JVM code for explicitly managing a free-list by the Camelot compiler
- A certificate on the heap space consumption of the program (the Isabelle proof fragment shown above).

It might be surprising to realise how little information needs to be encoded in this certificate to allow for a validation on the consumer side. The key step in achieving such a small certificate is the design of the heap space logic, tailored to the space inference on the high-level language. Since they share the overall structure, a single Isabelle tactic can be used for validation. This tactic uses the same path through the proof tree that was generated by the inference. In the case of recursive functions, the inferred heap bounds on the functions are used as invariants.

The infrastructure has to be secure against man-in-the-middle attacks, where the transmitted JAR file is intercepted and the certificate in it is replaced by a fake certificate. We achieve this by using a formally-grounded proof checker in the validation phase. If the fake certificate claims a resource bound that is too low, the proof checker will fail to prove the resource property with this certificate and hence execution will be prohibited. On the other hand, if the resource bound is fulfilled even for the fake certificate, validation will succeed and the code will be allowed to execute on the grounds that the bound in the certificate is provably met by the transmitted code. Thus, by only considering the behaviour of the transmitted program, rather than its authorship, we gain an unforgeable format of certificates.

C. Certificate validation:

To perform certificate validation on the consumer side, the Grail code is reconstructed. Furthermore, those components of the logical certificate that can be reconstructed out of the code and the transmitted certificate are reconstructed. The latter part contains the lemmas on the heap consumption, referring to the values in the Spec table shown above. A theory file combining these components is then passed to Isabelle/HOL to perform the check of the certificate.

Several auxiliary lemmas, for example establishing finiteness of the context, are proven first. In our example two main lemmas on the space consumption are proven next: one for the `ins` and one for the `sort` function. These lemmas state that the bodies of the functions `ins` and `sort` fulfil their specifications in the Spec table. A lemma on Context has to be proven, stating that this context is strong enough to prove all method, specification mappings contained in it. In particular, this means that the context contains entries for all sets of mutually recursive functions (see [35] for details). The main theorem that is proven in validating the certificate is `resourceStatement` below, which makes a statement of the overall heap consumption of the program, without referring to the extended types used in inference and heap space logic:

```
theorem resourceStatement:
 |> InsSort.init(INPUT) : initSpecDA
 ==> |> LET l1 =InsSort.init(INPUT)
        IN  InsSort.sort([l1]) END
          : % E h hh v p.
                ALL n.
                   h = emptyheap & E = emptyenv &
                   n = length INPUT -->
                   HSize hh < int 7
SUCCESS: Resource property proven
Elapsed time: 25.194sec
```

The code to be proven consists of an initialisation function `init`, that transfers the input data structure into the heap, which is specified as a resource assertion in `initSpecDA`. The body is a call to the function of interest: `InsSort.sort`. The assertion (after `%`, which represents a λ) states that if the execution starts with an empty heap and an empty environment, and if the input is a list of length n, then the heap consumption is strictly bounded by 7 heap cells. This upper bound 7 is specified on the consumer side. The proof of this overall bound directly builds on the proofs for the resource assertions of all functions, encoding resource bounds on their heap consumptions.

As a bigger example we can also certify and validate heap bounds on a heap-sort algorithm with in-place tree manipulation. The type inferred for this program is

```
HeapSort : 0, iList[0|int,#,0] -> iList[0|int,#,0], 0;
```

yielding the resource assertion $[\![l, 0, [l \mapsto \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!]$. As before, $\mathbf{L}(0)$ encodes the extended type of lists, with the weight 0 attached to the `Cons` constructor. Certificate checking is significantly more complex for this example. The use of non-empty dominator sets helps to reduce the checking time.

Table I summarises the data on the costs of certificate generation and validation. Our test machine was a 3.4GHz Intel Xeon, with 4GB main memory and 2MB cache. The proof checker is Isabelle 2003, run in batch mode, with Polyml 4.1.3, using a heap image that encodes all rules of the resource logic. We use a selection of list- and tree-based programs, named in the first column and sorted by bytecode size. The sizes of the generated bytecode and certificate, the latter compressed with gzip, are given in columns two and three, and the ratio in column four. We observe that for most of the programs, the size of the certificate is between 21% and 31%, with the latter value representing the most significant example, namely heap-sort. Only for the tiny example programs are the ratios higher. Column five lists the time for compilation and certificate generation. It is never higher than about half a second. This demonstrates the strength of our infrastructure in producing small certificates with low generation overhead. The final two columns specify the time (in seconds) and memory footprint (in MB of resident memory) needed at the consumer side for validating the certificate. Here the overhead is much higher because we use an entire theorem prover to perform validation. This adds high constant overhead, as can be seen by the memory footprint, which is about 20MB even for small programs. For the heap-sort example we observe

TABLE I
RUNTIME AND MEMORY FOOTPRINT OF CERTIFICATE GENERATION AND VALIDATION

Program	Code	Certificate	Cert size/	Cert. gen.	Validation	
	size (kB)	size (kB)	Code size	time (sec)	time (sec)	memory (MB)
Clone	0.67	0.54	80.7%	0.03	1.5	21
Mirror	0.87	0.56	64.2%	0.05	2.5	20
Flatten	1.11	0.68	61.2%	0.06	6.0	22
RevApp	2.62	0.57	21.8%	0.12	2.4	23
Twice	2.75	0.64	23.2%	0.13	4.0	26
EvenOdd	2.81	0.62	22.1%	0.13	4.5	24
InsSort	2.87	0.74	25.8%	0.14	9.7	26
Nub	2.95	0.76	25.9%	0.17	6.9	69
Merge	3.32	0.76	22.8%	0.22	11.4	76
HeapSort	4.81	1.49	31.0%	0.52	135.7	190

ca. 2 minutes validation time, consuming 190MB. While such high costs for validation are a major weakness of our current implementation, they are not inherent to our approach. Since we do not use automated proof search during validation, there is no fundamental obstacle in implementing a stand-alone proof checker. This would drastically reduce the costs on the consumer side and further improve the usefulness of our infrastructure. Also, our main application domain is distributed systems, with powerful machines on code consumer side. For embedded systems we do not expect to support on-device validation, and rather delegate the off-line validation process to a third party machine.

An on-line demo of our PCC infrastructure, together with Camelot example programs and a set of exercises can be found at: `http://projects.tcs.ifi.lmu.de/mrg/demo.php`

V. Summary and Future Work

Our PCC infrastructure achieves a high degree of dependability of distributed systems with mobile code, by attaching certificates, in the form of condensed formal proofs of bounded resource consumption, to the JVM code that is transmitted. By this foundations-driven design, we gain an unforgeable format of certificates, we are not relying on a trust relationship between producer and consumer, and we achieve a scalable system without a centralised verification agency.

Our complete PCC infrastructure contains a certifying compiler, which automatically infers linear upper bounds on the heap consumption of programs in a strict, first-order functional language and generates certificates for bounded heap consumption. These certificates are then independently and automatically checked on the consumer side before executing the program. This infrastructure greatly increases the dependability in large distributed networks, by preventing mobile code from exhausting available resources. Our technique of embedding certificates in a theorem prover is flexible enough to allow a manual generation of certificates where they cannot be generated automatically (if for example the heap consumption is quadratic) or for other forms of resources.

In continuation of our work on resource analysis, a type system that encodes heap space consumption for a small object-oriented language RAJA has been developed [26] and type checking has been implemented [37]. Refinements of the resource analysis for predicting stack space have been developed for a Camelot-like language [25].

Several projects build on the techniques presented here to apply them to real-world applications. The most ambitious of these is the Mobius project [38], which aims to provide safety and security guarantees for Java-enabled mobile devices. This project has studied a number of foundational PCC techniques, including a formalisation of the JVM bytecode language in the Coq proof assistant [39]; type systems for resource consumption and information flow, and their integration with program logics [38]; and a technique whereby provably correct on-device checkers are extracted from Coq developments of various static analyses [40]. In the context of Grid computation the ReQueST project [41] focused on scientific applications involving large databases, which are too large for users to have individual copies: instead, mobile programs are used to perform analyses on central databases. The project has addressed the problem of certifying time and space requirements, using Java bytecode as source language. The logical basis has been an executable Coq formalisation of the JVM [42], obtained through automated code extraction. In the context of embedded systems the EmBounded project [43] has extended our resource analysis to the expression layer of Hume, a strict, higher-order language with algebraic data types. For concrete embedded systems control and computer vision applications, this analysis infers good bounds on heap, stack and worst-case execution time [44].

Acknowledgements

We would like to thank our colleagues in the MRG project, in particular Don Sannella, Martin Hofmann, David Aspinall, Robert Atkey, Brian Campbell, Stephen Gilmore, Alberto Momigliano, Olha Shkaravska, and Ian Stark. The research presented here was supported by the EU as projects IST-2001-33149 (MRG) and IST-2005-15905 (Mobius), under the FET proactive initiative on Global Computing, and as project IST-510255 (EmBounded), under the Open-FET programme. It was also partly funded by the EPSRC under project EP/C537068/1 (Request), and by the EU under the Framework 6 grant RII3-CT-2005-026133 (SCIEnce).

REFERENCES

[1] G. Necula, "Proof-carrying-code," in *Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, January 15–17, 1997, pp. 106–116.

[2] M. Hofmann and S. Jost, "Static Prediction of Heap Space Usage for First-order Functional Programs," in *Symposium on Principles of Programming Languages (POPL'03)*. New Orleans, USA: ACM Press, Jan 2003, pp. 185–197.

[3] G. Necula, "Compiling with Proofs," Ph.D. dissertation, Carnegie Mellon University, Sep. 1998. [Online]. Available: http://raw.cs.berkeley.edu/Thesis/thesis.pdf

[4] G. Necula and P. Lee, "Safe Kernel Extensions without Run-time Checking," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. SI, pp. 229–243, 1996.

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

[6] G. Necula and P. Lee, "Efficient Representation and Validation of Proofs," in *Symposium on Logic in Computer Science (LICS'98)*. Indianapolis, USA, 21–24 June: IEEE Computer Society, 1998, pp. 93–104.

[7] ——, "Safe, Untrusted Agents Using Proof-carrying-code," in *Special Issue on Mobile Agent Security*, ser. LNCS, vol. 1419. Springer, 1998, pp. 61–91.

[8] A. Appel, "Foundational Proof-Carrying Code," in *Symposium on Logic in Computer Science (LICS'01)*. IEEE Computer Society, Jun. 2001, pp. 247–258.

[9] D. Wu, "Interfacing Compilers, Proof Checkers, and Proofs for Foundational Proof-carrying code," Ph.D. dissertation, Princeton University, 2005.

[10] A. Appel, N. Michael, A. Stump, and R. Virga, "A Trustworthy Proof Checker," *J. Autom. Reason.*, vol. 31, no. 3–4, pp. 231–260, 2003.

[11] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo, "An Open Framework for Foundational Proof-carrying-code," in *Types in Language Design and Implementation (TLDI'07)*. Nice, France: ACM Press, January 2007, pp. 67–78.

[12] N. Hamid, Zhong Shao, V. Trifonov, S. Monnier, and Zhaozhong Ni, "A Syntactic Approach to Foundational Proof-carrying code," in *Symposium on Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, July 22–25, 2002, pp. 89–100.

[13] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo, "Foundational Typed Assembly Language with Certified Garbage Collection," in *Symposium on Theoretical Aspects of Software Engineering (TASE'07)*. IEEE Computer Society, 2007, pp. 326–338.

[14] M. Wildmoser and T. Nipkow, "Asserting Bytecode Safety," in *European Symposium on Programming (ESOP'05)*, ser. LNCS, vol. 3444. Edinburgh, UK, April 4–8: Springer, 2005, pp. 326–341.

[15] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz, "Prototyping Proof-carrying-code," in *Exploring New Frontiers of Theoretical Informatics*, 2004, pp. 333–347.

[16] E. Albert, G. Puebla, and M. Hermenegildo, "Abstraction-Carrying Code," in *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, ser. LNCS, vol. 3452. Montevideo, Uruguay, March 14–18: Springer, 2005, pp. 380–397.

[17] B. Wegbreit, "Mechanical Program Analysis," *Commun. ACM*, vol. 18, no. 9, pp. 528–539, 1975.

[18] D. Le Métayer, "ACE: an Automatic Complexity Evaluator," *ACM Trans. on Program. Lang. Syst*, vol. 10, no. 2, pp. 248–266, 1988.

[19] M. Rosendahl, "Automatic Complexity Analysis," in *Intl Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. Imperial College, London, UK: ACM, 1989, pp. 144–156.

[20] S. Debray and N.-W. Lin, "Cost Analysis of Logic Programs," *ACM Trans. on Program. Lang. Syst.*, vol. 15, no. 5, pp. 826–875, 1993.

[21] R. Wilhelm et al, "The Worst-case Execution-time problem — Overview of Methods and Survey of Tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[22] J. Hughes, L. Pareto, and A. Sabry, "Proving the Correctness of Reactive Systems Using Sized Types," in *Symposium on Principles of Programming Languages (POPL'96)*. St. Petersburg Beach, USA: ACM, 1996, pp. 410–423.

[23] M. Hofmann, "Linear Types and Non-size-increasing Polynomial Time Computation," *Inf. Comput.*, vol. 183, no. 1, pp. 57–85, 2003.

[24] D. Aspinall and M. Hofmann, "Another Type System for In-place Update," in *European Symposium on Programming Languages and Systems (ESOP'02)*, ser. LNCS, vol. 2305. Grenoble, France, April 8–12: Springer, 2002, pp. 36–52.

[25] B. Campbell, "Amortised Memory Analysis Using the Depth of Data Structures," in *European Symposium on Programming Languages and Systems (ESOP'09)*, ser. LNCS, vol. 5502. York, UK, March 22–29: Springer, 2009, pp. 190–204.

[26] M. Hofmann and S. Jost, "Type-based Amortised Heap-space Analysis," in *European Symposium on Programming (ESOP'06)*, ser. LNCS, vol. 3924. Vienna, Austria, March 27–28: Springer, 2006, pp. 22–37.

[27] O. Shkaravska, R. van Kesteren, and M. van Eekelen, "Polynomial Size Analysis of First-order Functions," in *Typed Lambda Calculi and Applications (TLCA'07)*, ser. LNCS, vol. 4583, Paris, June 26–28, 2007, pp. 351–365.

[28] V. Braberman, D. Garbervetsky, and S. Yovine, "A Static Analysis for Synthesizing Parametric Specifications of Dynamic Memory Consumption," *Journal of Object Technology*, vol. 5, no. 5, pp. 31–58, Jun. 2006.

[29] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine, "Parametric Prediction of Heap Memory Requirements," in *International Symposium on Memory Management (ISMM'08)*, ser. SIGPLAN. Tucson, USA, June 7–8: ACM, Jun. 2008, pp. 141–150.

[30] K. MacKenzie and N. Wolverson, "Camelot and Grail: Resource-aware Functional Programming on the JVM," in *Trends in Functional Programing*, vol. 4. Edinburgh, UK: Intellect, Sep. 2003, pp. 29–46.

[31] L. Beringer, K. MacKenzie, and I. Stark, "Grail: a Functional Form for Imperative Mobile Code," *Electronic Notes in Theoretical Computer Science*, vol. 85, pp. 3–23, Jun. 2003.

[32] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska, "Automatic Certification of Heap Consumption," in *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)*, ser. LNCS, vol. 3452. Montevideo, March 14–18: Springer, Feb 2005, pp. 347–362.

[33] M. Berkelaar. lp_solve. Eindhoven Univ of Technology. An LP-solver released under the Lesser GNU public licence. Version 4.0.1.0. [Online]. Available: ftp://ftp.es.ele.tue.nl/pub/lp_solve

[34] R. Tarjan, "Amortized Computational Complexity," *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, no. 2, pp. 306–318, April 1985.

[35] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano, "A Program Logic for Resources," *Theoretical Computer Science*, vol. 389, no. 3, pp. 411–445, Dec. 2007.

[36] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[37] M. Hofmann and D. Rodriguez, "Efficient Type-checking for Amortised Heap-space Analysis," in *Conference on Computer Science Logic (CSL'09)*, ser. LNCS. Coimbra, Portugal, Sep 7–11: Springer, 2009, to be published.

[38] MOBIUS. Consortium, "Project deliverables," 2009. [Online]. Available: http://mobius.inria.fr

[39] D. Pichardie, "Bicolano — Byte Code Language in Coq," 2006, Summary appears in [38], Deliverable 3.1. [Online]. Available: http://mobius.inria.fr/bicolano

[40] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu, "Extracting a Data Flow Analyser in Constructive Logic," *Theoretical Computer Science*, vol. 342, no. 1, pp. 56–78, Sep 2005.

[41] ReQueST. Consortium, "Project deliverables," 2009. [Online]. Available: http://groups.inf.ed.ac.uk/request/

[42] R. Atkey, "CoqJVM: An Executable Specification of the Java Virtual Machine Using Dependent Types," in *Types for Proofs & Programs (TYPES'07)*, ser. LNCS, vol. 4941, Cividale des Friuli, Italy, May 2–5, 2007, pp. 18–32.

[43] EmBounded. Consortium, "Project deliverables," 2009. [Online]. Available: http://www.embounded.org/

[44] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann, ""Carbon Credits" for Resource-bounded Computations Using Amortised Analysis," in *Intl. Symp. on Formal Methods (FM'09)*, Eindhoven, The Netherlands, November 2–6, 2009, submitted for publication.