
The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006



LICS'06 Workshop

PCC 2006:

International Workshop on Proof Carrying Code

August 11, 2006

Proceedings

Editors:

Adriana Compagnoni and Amy Felty

Contents

Keynote Addresses

Andrew Appel , A Very Modal Model of a Modern, Major, General Type System	1
Ian Stark , Resource Guarantees and PCC: 50 ways to say it with a proof	2

Invited Talks

Ricardo Medel , Information Flow Analysis for Low-Level Languages	3
Tamara Rezk , Certificate Translation	4
Zhong Shao , A Translation from Typed Assembly Languages to Certified Assembly Programming	5
Dachuan Yu , Toward More Typed Assembly Languages for Confidentiality	6

Poster Abstracts

Robert Atkey and Kenneth MacKenzie , ReQueST: Resource Quantification for e-Science Technologies	7
Adam Chlipala , Developing Certified Program Verifiers with a Proof Assistant	10
João Gomes, Daniel Martins, Simão Melo de Sousa, and Jorge Sousa Pinto , LISSOM, a Source Level Proof Carrying Code Platform	13

A Very Modal Model of a Modern, Major, General Type System: Keynote Address

Andrew Appel
Princeton University
appel@princeton.edu

Semantic approaches to machine-checkable proofs of type soundness for Proof-Carrying Code have not been easy—until now. We present a clean and powerful model of recursive and impredicatively quantified types with mutable references. Our model is less restrictive than previous models of impredicative references. We model all of the type constructors needed for typed intermediate languages and typed assembly languages for object-oriented and functional languages; our technique is applicable to any small-step semantics including lambda-calculus, labeled transition systems, and von Neumann machines. We have reduced the system to such a simple formulation that it becomes clear that what we have is a Kripke semantics of the Goedel-Loeb (GL) logic of provability. We have machine-checked proofs in Coq.

This is joint work with Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon.

Resource Guarantees and PCC: 50 ways to say it with a proof:¹ Keynote Address

Ian Stark
The University of Edinburgh
Ian.Stark@ed.ac.uk

In collaboration with various European partners, the Mobility+Security group at Edinburgh (see <http://www.lfcs.ed.ac.uk/m+s>) have been investigating the use of Proof-Carrying Code to implement *Resource Guarantees* for the Java Virtual Machine.

I'll describe and demonstrate a working implementation of this, where Java class files carry certificates of their heap space usage that are independently verified before execution. These certificates are generated during compilation of an ML-like source language into Java bytecodes; using resource types, inferred from the source program, that capture the required information about heap usage. Code consumers may specify resource policies against which incoming class files are checked; where both certificates and policies are expressed in a general bytecode logic that can capture various different kinds of resource.

More generally, I'll discuss associated work on some of the very many different ways to engage PCC: such as wholesale code certification for application distributors; tactic-carrying code; algebras of resources; validation of optimising compilation; adaptive proofs for heterogeneous clients; languages for resource policies; probabilistically checkable proofs; and PCC in large scientific databases.

This work is part of the MRG, ReQueST and Mobius projects funded by the UK EPSRC and the European Commission Framework Programme.

¹Contents may vary.

Information Flow Analysis for Low-Level Languages: Invited Talk

Ricardo Medel
Stevens Institute of Technology
rmedel@cs.stevens.edu

This work addresses the problems raised by the study of confidentiality for Multi-Level Security systems in the context of low-level languages. In particular, we studied the vulnerabilities introduced by the absence of control flow constructs, the reuse of registers, and the manipulation of a control stack. In this talk, I will present language-based techniques used for the development of type systems for low-level languages that guarantee that well-typed programs preserve confidentiality.

Certificate Translation: Invited Talk

Tamara Rezk
INRIA Sophia-Antipolis
Tamara.Rezk@sophia.inria.fr

Program verification techniques based on programming logics and verification condition generators provide a powerful means to reason about programs. Whereas these techniques have very often been employed in the context of high-level languages in order to benefit from their structural nature, it is often required, especially in the context of mobile code, to prove the correctness of compiled programs. Thus it is highly desirable to have a means of bringing the benefits of source code verification to code consumers.

We propose certificate translation, a mechanism that allows to transfer evidence from source programs to compiled programs. It builds upon the notion of certificate, which is used in Proof Carrying Code architectures to convey to the code consumer easily verifiable evidence that programs respect some policy. A certificate translator is an algorithm that turns certificates of source programs into certificates of compiled programs; its definition is tightly bound to the compiler used for programs, and the main difficulty in defining one stems from the optimizations performed by the compiler. In order to illustrate the feasibility of our approach, we build certificate translators for an optimizing compiler from a simple imperative language to an intermediate RTL language.

A Translation from Typed Assembly Languages to Certified Assembly Programming: Invited Talk

Zhong Shao
Yale University
shao-zhong@cs.yale.edu

Typed assembly languages (TAL) and certified assembly programming (CAP) are two new techniques for certifying assembly programs and for building proof-carrying code. TAL uses syntactic types (for reasoning) while CAP uses Hoare-style logic assertions. Because of the differences between type system and logic, TAL and CAP are suitable for different kinds of programming tasks. Previously, programs verified in TAL and CAP can not directly interoperate with each other so it is hard to integrate them into a single PCC system. In this talk we compare the TAL and CAP approaches and present a type-preserving translation from TAL to XCAP—a recent CAP language that provides modular support for embedded code pointers and impredicative polymorphism. Our translation supports other common language features such as general references and recursive types. Our paper gives a clear account of the relationship between type-based and logic-based approaches for certifying assembly programs. This is a joint work with Zhaozhong Ni at Yale University.

Toward More Typed Assembly Languages for Confidentiality: Invited Talk

Dachuan Yu
Research Engineer
DoCoMo USA Labs
yu@docomolabs-usa.com

Low-level languages are starting to receive attention in the studies of information-flow questions, especially from the perspective of typed assembly languages and certifying compilation. In some recent work, type annotations are used to restore the missing abstractions in assembly code, and type-preserving compilation is used to preserve security evidence from the source to the target. By security-type checking directly at the target level, code consumers gain higher confidence on the behavior of untrusted code. Although a good start, existing work in this area has only focused on relatively simple settings, largely ignoring the timing behaviors of assembly programs. Unfortunately, timing behaviors, both external and internal ones, present some channels of information flow that can be easily exploited.

This talk presents our progress toward addressing some of these covert information-flow channels in assembly code, including termination, timing, possibilistic and probabilistic channels. It is inspired by source-level information-flow type systems which account for program timing behaviors. On top of our previous work on a typed assembly language for confidentiality, we introduce additional annotations to document the timing of the program execution. These timing annotations are used to express various source-level constraints, such as the absence of high loops in any security contexts and the absence of any loops in high security contexts. We discuss how to accommodate these annotations in a type system and how to produce these annotations given securely typed source programs.

ReQueST: Resource Quantification for e-Science Technologies

Robert Atkey and Kenneth MacKenzie
LFCS, School of Informatics, University of Edinburgh
bob.atkey@ed.ac.uk, kwxm@inf.ed.ac.uk

17th July 2006

1 e-Science and the Grid

Increasingly, large scale science is being carried out through distributed global collaborations enabled by the Internet. Computational grids [4] provide powerful computational and data storage services via large-scale networks. Many areas of modern science, such as astronomy, particle physics, chemistry, biology and healthcare now have databases whose size is measured in terabytes or petabytes and the Grid provides an ideal means for accessing and processing such data. Some examples demonstrating the scale of current e-Science activities include:

- In 2007 the Large Hadron Collider (LHC) will come into service at CERN in Geneva. This will study particle collisions at very high energies. It is estimated that LHC will produce 15 petabytes (15 million gigabytes) of data per year, all of which will need to be stored, shared and analysed.
- In the USA, the Sloan Digital Sky Survey (SDSS) is currently in progress. This aims to provide detailed optical images covering more than a quarter of the sky, and a 3-dimensional map of about a million galaxies and quasars. Later surveys will collect specialised information to address fundamental questions about the nature of the Universe, the origin of galaxies and quasars, and the formation and evolution of our own Galaxy. The data which has been collected for the basic sky survey (and which can be accessed at the SDSS SkyServer website cas.sdss.org) already exceeds 14 terabytes. Data from the SDSS and other astronomical databases is also accessible via web services at the Virtual Observatory Web Services site, www.voservices.org.
- The IXI (Information eXtraction from Images) project uses the Grid to enable analysis of large amounts of medical imaging data. One aim is to help research on rheumatoid arthritis via extraction of information from magnetic resonance images. The image analysis algorithms involved can take several hours per image, so analysis of large quantities of images can take a very considerable time. Grid computation can speed this process up by distributing image analysis tasks to many different computers.

2 Resource Guarantees and PCC for the Grid

Large-scale e-Science projects can generate so much data that it is impractical to take the traditional route where each scientist keeps a local copy of the data on their computer for analysis. Instead, users will have to send programs across the network to be executed on or near the database itself. When the machines involved are shared between many users it is imperative that one user's program does not monopolise all the resources of the machine. The current mechanisms for resource specification on the Grid are somewhat

crude. A user may supply estimates of execution time and memory requirements which are then used for scheduling jobs, but these estimates are usually just informed guesses, with no guarantee that they are accurate. This can lead to situations where a user submits a job which fails to complete due to lack of memory, for example: in this case the job will simply be terminated. The user (or their funding body) may have paid real money for access to the remote computer, and this money will have been wasted.

The ReQueST approach aims to avoid this type of problem. We hope to use advanced static analyses and Proof-Carrying Code to provide accurate and certified resource bounds for Grid applications. We distinguish between two different kinds of Grid application in current use: *compute grids*, where a user submits a piece of code and its data for processing somewhere on the Grid; and *data grids*, where large databases such as the ones mentioned earlier are made accessible via wide-area networks. Both of these applications will benefit from more information about resource usage, but we believe that in the immediate future data grids will have the most to gain from PCC. Current compute grids are usually comprised of clusters of nodes isolated from the network and arranged so that they may be easily reset to a known state, lessening the impact of malicious or badly-written code.

Data grids such as the Virtual Observatory allow remote users to submit queries against the data in order to retrieve subsets for local analysis. Such queries allow the selection of all objects within a region of the sky or all objects of a certain type, but more complex queries, involving detailed analysis of spectra for example, are limited by the capabilities of SQL. Most modern database server systems allow the installation of *stored procedures* (also known as *user defined functions*). Stored procedures are pieces of code that are executed within the database process and are used to augment the capabilities of the query language. Traditionally they have been written in proprietary extensions of SQL, but now they can be written in high-level languages such as Java or C#. As an example, a new function for analysing photographs taken by a telescope may be installed and subsequently used in normal SQL queries. Performing such analysis on the server reduces the amount of data that must be transferred. Further applications include custom federation of databases, where users submit queries on other remote databases from within queries.

The prospect of running untrusted code that can possibly access valuable resources such as the network from within the database process itself is an ideal application for proof-carrying code. Database server software already dynamically prevents stored procedures from accessing the local disk or using the network, but does not constrain their resource usage. We will use a certifying compiler which not only produces a compiled version of the source program, but also uses the results of static analyses to calculate bounds for the resource consumption of the program and produce a condensed proof that the program satisfies these bounds; the database server can then check the proof before executing the stored procedure.

Ultimately, it is likely that the two kinds of grid will converge, and PCC will have a role to play in complex federations of computing and data resources.

In the earlier Mobile Resource Guarantees project [5] we developed cost models and proof techniques for Java bytecode and a PCC system based on bytecode generated from a high-level functional language called Camelot. The ReQueST project is adapting these techniques for application to Grid programs; we aim to extend to Java as a high-level language and introduce techniques for handling external library code (for instance, compiled from C or Fortran) whose source is not available.

3 Contributions

Static analysis of resource usage. We wish to provide an automated framework which integrates well with established programming practice, and to this end we aim to use the ESC/Java2 [3] static analysis tool as a resource bound analyser. ESC/Java2 accepts Java programs which have been augmented with Java Modelling Language (JML) specifications of program behaviour, and performs automated static checking

for violation of these specifications. JML already has some support for annotations specifying time and space usage, but these are somewhat limited. We have already made some progress in improving and extending the JML resource-related annotations and modifying ESC/Java so that it can verify specified bounds on heap space allocation. Further details appear in [2].

ESC/Java2 does not produce independently verifiable proofs (and indeed is unsound) so it is not suitable for PCC. However, with more work, it will be useful for verifying heap space bounds on users' jobs before they are submitted to compute grids, ensuring that the resource allocation they have requested is sufficient. We are currently investigating the use of the Coq proof assistant for formalising Java bytecode to provide a sound, verified basis for PCC.

Resource policies. In [1] we have investigated resource specifications and policies. In our scenario, a code producer provides per-method resource bounds which are functions of the sizes of input arguments. Meanwhile, the code consumer has a policy which provides a promise that method arguments shall not exceed certain sizes, but requires in turn that the method's resource consumption shall not exceed a certain constant. The form of our policies means that it is easy for a consumer to test whether its resource policy is satisfied provided the producer's claimed bounds actually hold. If the test fails then a job can be rejected immediately; if the test succeeds then the consumer can go on to check a proof that the producer's code actually satisfies the given bounding function. Our policies can be made parametric in platform-dependent library functions, so that it is unnecessary for the producer to know the precise resource behaviour of a target platform in advance. See [1] for more details.

An application to Astronomy. We are currently collaborating with researchers at the Royal Observatory, Edinburgh. We are investigating the possibility of using static analysis and PCC to certify resource properties of stored procedures running on astronomical databases.

Acknowledgment. This work was funded by the ReQueST grant (EP/C537068) from the Engineering and Physical Sciences Research Council.

References

- [1] David Aspinall and Kenneth MacKenzie. Mobile resource guarantees and policies. In *Proc. Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2005)* (LNCS 3956), pages 16–36. Springer-Verlag, 2006.
- [2] Robert Atkey. Specifying and verifying heap space allocation with JML and ESC/Java2. In 8th Workshop for Formal Techniques for Java-like Programs (FTfJP2006), Nantes, July 2006.
- [3] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128, January 2005.
- [4] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [5] Donald Sannella, Martin Hofmann, David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Hans-Wolfgang Loidl, Kenneth MacKenzie, Alberto Momigliano, and Olha Shkaravska. Mobile Resource Guarantees. In *Trends in Functional Programming*, volume 6. Intellect, September 2005.

Developing Certified Program Verifiers with a Proof Assistant

Adam Chlipala
Computer Science Division
University of California, Berkeley
Berkeley, California, USA
adamc@cs.berkeley.edu

Abstract

I describe ongoing work on a new approach to foundational proof-carrying code. The key new idea is to use *certified program verifiers* to embody customized program verification strategies, specialized to particular safety policies, enforcement mechanisms, and source-level compilers. A certified verifier is an executable program that has a full correctness proof.

The particular strategy that I've been following involves using the Coq computer proof assistant as an environment for dependently typed programming, where types ensure total correctness. Elements of the development are interesting for the general insight they provide into programming with specifications.

1 Introduction

This poster describes my work on implementing program verification tools that have formal soundness proofs, using the Coq proof assistant [BC04].

The idea of certified program verifiers has important practical ramifications for foundational proof-carrying code (FPCC) [App01]. Like traditional proof-carrying code (PCC), FPCC is primarily a technique for allowing software consumers to obtain strong formal guarantees about programs before running them. It differs in that its trusted base is independent of particular source languages and compilation strategies. In this way, it gains both generality and higher assurance of soundness over the first PCC systems.

The germ of the project I'll describe comes from past work on improving the runtime efficiency of FPCC program checking. Perhaps the largest obstacle to practical use of FPCC stems from the delicate trade-offs between generality on one hand and space and time efficiency of proofs and proof checkers on the other. Program verifiers like the Java bytecode verifier have managed to creep into wide use almost unnoticed by laypeople, but naive FPCC proofs are much larger than the metadata included with Java class files and take much longer to check. It's unlikely that this increased burden would be acceptable to the average computer user.

Fundamentally, custom program verifiers with specialized algorithms and data structures have a leg up on very general proof-based verifiers. In our initial work on certified program verifiers [CCN06], we proposed getting the best of both worlds by moving up a level of abstraction: allow developers to ship their software with specialized *proof-carrying verifiers*. These verifiers have the semantic functionality of traditional program verifiers and model-checkers, but they also come with machine-checkable proofs of soundness. Each such proof can be checked *once* when a certified verifier is installed. After the proof checks out, the verifier can be applied to any number of similar programs. These later verifications require no runtime generation or checking of uniform proof objects, which we found to be the major bottleneck in previous experience with FPCC. Our paper presents performance results showing an order of magnitude improvement over all published verification time figures for FPCC systems for Typed Assembly Language [MWCG99] programs, by using a certified verifier. The verifier had a complete soundness proof, so no formal guarantees were sacrificed to win this performance.

2 Implementing Certified Verifiers with Coq

The main problem that we encountered was in the engineering issues of proof construction. We used a more or less traditional approach to program verification in proving the soundness of our verifiers, writing them in a standard programming language and extracting verification conditions that imply their soundness. Keeping the proof developments in sync with changes to verifier source code was quite a hassle. We also found that the structure of the verifier program and its proof were often very closely related, leading to what felt like duplicate work.

I decided to try investigating what could be gained by writing verifiers from the start in a language expressive enough to encode verifier soundness in its type system. My upcoming paper [Chl06] describes a case study using Coq to develop a complete memory safety verifier for x86 machine code programs that use ML-style algebraic datatypes.

My approach combines explicit dependently-typed programming with interactive and automated theorem proving inside of Coq. By providing decision procedures and other verifier ingredients with types that only admit sound implementations, it's possible to avoid implicit dependencies between programs and proofs about them. This style of programming is not new, but my results show that it can be seen through to completion in a reasonable amount of time.

My implementation uses a number of Coq type families in the style of refinement types [FP91], including some standard ones and some new ones that I define. Refinement types refine type systems by providing types that describe, for instance, “every value of type τ that satisfies logical predicate P .” Standard refinement types stick to predicates that facilitate effective type inference, while Coq provides the mechanisms to use arbitrary logical predicates. Proper use of values of refinement types is validated through explicit construction of proof terms in programs.

I use types like $\{\{x : \tau \mid P(x)\}\}$, which (for an arbitrary logical predicate P) is an option-like type whose values are either special failure values or packages containing a value x of type τ and a proof that $P(x)$ holds. These type families can be treated as failure monads, leading to a style familiar from Haskell programming. Computations can proceed with sequences of calls to potentially-failing subroutines, where each successful call binds a return value and a proof of its correctness for use in later calls. Monadic syntax makes the code as clear visually as solutions that use, e.g., exceptions.

Coq's features for semi-automated proof construction and program extraction work very well with this style. It's possible to write programs with “proof holes” that are queued as goals for the user to prove interactively, which is a whole lot nicer than writing out proof terms manually. Program extraction also provides for a compilation from a finished Coq development into OCaml code, which can then be compiled to speedy native code. Extraction erases all proof-related parts of the program, and the Coq type system guarantees that this erasure is semantics-preserving for well-typed terms. Thus, total correctness theorems about an original Coq verifier are guaranteed to hold about the compiled version that we actually run, modulo bugs in Coq and OCaml.

The other essential technique is the use of Coq's module system to build verifiers from reusable components. Functors serve as a tool for translating a verifier at one level of abstraction into a verifier for a lower level. In this way, my final implementation is based on a stack of 8 abstraction levels, from x86 machine code to a high-level, declarative type system description. Like at the level of individual functions, dependent types make it relatively painless to glue components together, since there's no need to stare at the final product and devise a customized, global proof strategy. The module system ensures that the proof parts of the different components work smoothly together.

3 Conclusion

In the history of both traditional and foundational PCC, research has started out focusing on expressivity and the basic concepts. From there, the next steps involve techniques to improve algorithmic efficiency. Especially for FPCC, there is a need for a third stage focusing on practical issues in the development of new FPCC pieces. We shouldn't rest satisfied as long as the difficulty of implementing support for a new compiler within an FPCC framework is measured in PhD theses. I hope that the kinds of software engineering techniques that I've mentioned here can provide some insight into the challenges of this third stage.

References

- [App01] Andrew W. Appel. Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, 2001.
- [BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [CCN06] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *VMCAI '06: Proceedings of the Seventh International Conference on Verification, Model Checking and Abstract Interpretation*, January 2006.
- [Chl06] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Conference record of the 11th ACM SIGPLAN International Conference on Functional Programming*, September 2006. To appear.
- [FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.
- [MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

LISSOM, a Source Level Proof Carrying Code Platform

João Gomes	Daniel Martins	Simão Melo de Sousa	Jorge Sousa Pinto
	Departamento de Informática		DI/CCTC
	Universidade da Beira Interior		Universidade do Minho
	Covilhã, Portugal		Braga, Portugal
	{joao.gomes,daniel.martins,desousa}@di.ubi.pt		jsp@di.uminho.pt

Abstract

This paper introduces a proposal for a Proof Carrying Code (PCC) architecture called LISSOM. Started as a challenge for final year Computing students, LISSOM was thought as a mean to prove to a sceptic community, and in particular to students, that formal verification tools can be put to practice in a realistic environment, and be used to solve complex and concrete problems. The attractiveness of the problems that PCC addresses has already brought students to show interest in this project.

1 The Lissom Platform

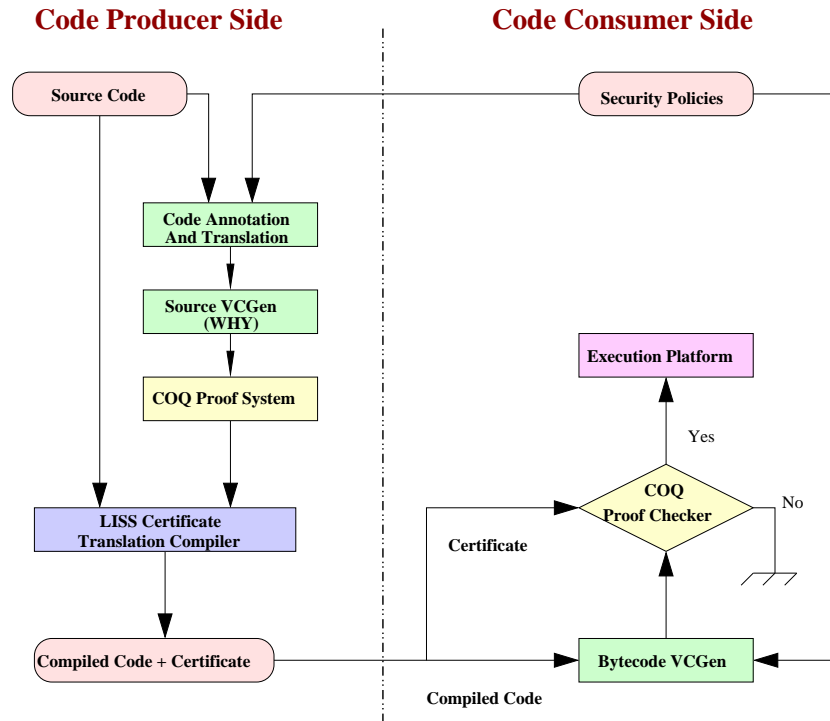
Traditional PCC architectures center their certificate generation mechanisms on the output of the compilation. Along the lines of recent projects, we believe that there are strong benefits in moving the certificate generation to the source code level. Because there exist good tools for source code verification and for formal verification in general, it is a feature of the LISSOM platform that existing tools are used as much as possible at key points of its infrastructure.

Our vision of PCC is based on the following two underlying principles:

- *Source level PCC is the way.* It is our belief that the realistic formal verification of mobile code should be performed at source level. Programmers may be unaware of the target architecture details, and in general algorithmic constructions are expressed at source level.
- *Reuse as much as possible.* There exist plenty of powerful tools for the formal verification of source code. Such tools already have experienced user communities, and have reached an appreciable level of maturity and flexibility that make them natural choices in the context of a source level PCC architecture.

Our main goal with LISSOM is to get experienced with PCC and to put it into practice. We now describe the proposed architecture for the LISSOM platform (see figure below).

The Source Language and the Compiler. LISS (Language for Integers Sets and Sequences) is a non-trivial toy language that also features a realistic type system (with e.g. sets, vectors a la Java, etc.) and high-level constructs. LISS is intended here as a suitable test-bed before aiming at an industrial-level language. This language must be extended in order to provide an annotation system for the source code. In a source level PCC architecture, the compiler has to compile source code but also proofs into their machine level counterpart. A very interesting challenge is to transform a source level structural proof (proofs heavily rely on the structure of the analyzed program) into a proof that is still structurally close to the machine level code. We follow here the contributions of [1]).



The Virtual Machine. LISSOM uses a sequential, stack-based virtual machine, which despite its simplicity has the capacity to support real languages (such as C or Java), and is, together with LISS, a suitable test-bed. The machine is an adaptation of Filliatre’s original virtual machine [4], used for teaching several courses at our universities (e.g. Compiler Construction and Formal Methods).

The Proof System and the Proof Checker. As far as the *Trusted Computing Base* (TCB) is concerned, it is important for it to be as small and solid as possible; we believe that an adequate choice of proof system may help attaining such a goal. Also, it is important to be able to express high level policies as well as lower level ones.

These requirements have led us to consider using the COQ proof system and its higher-order specification language and underlying proof mechanisms. This system, based on the calculus of inductive constructions, has been used with success for the formal verification of critical and large-scale systems. As far as source code is concerned, integration with COQ is guaranteed by the existence of a number of tools suited for code annotation and proof (e.g. [6]). LISSOM will feature a source code verification system based on WHY and the COQ system. Thus we intend to use COQ proof objects as certificates.

The Verification Generator. This will be obtained using Filliatre’s WHY tool [5], which is capable of producing proof obligations for various systems, including COQ. We are presently working on a WHY module for the language LISS (equivalent to *Caduceus* for C [6]) and for the input language of the virtual machine. The annotation language used for this will be an adaptation of JML [3] specialized for the security policy specification.

2 Road Map

After the present prototyping phase, the platform must have proved to be adequate for mobile code security. The relevance and conceptual solidity of previous works on formal verification (e.g. [2]) on which this is based lead us to believe in the success of the enterprise. Our road-map is the following, where the points highlighted as *in progress* are the modules which are currently in active development.

1. To design an annotation language for LISS (in progress);
2. to extend the WHY tool to contemplate this annotation language, allowing to use WHY as a generator of proof obligations for source code (in progress);
3. to design a proof system for the LISS language, integrated in COQ (starting phase);
4. to extend the LISS compiler with the capability to translate certificates (starting phase);
5. to design a proof system for the virtual machine and its language, integrated in COQ (in progress);
6. to design a proof obligation generator for compiled code (in progress);

We finish with a few examples of the many interesting problems that will be raised in this work, along with the classical challenges that every PCC platform must address. A first problem is the automation of the COQ proof process and its impact on the consumer effort; the tension between expressiveness and automation is a well-known problem that must be carefully studied. It also remains to see to what extent the techniques presented in [7] allow for conciseness of COQ certificates.

The language LISS and the virtual machine used are non-trivial, but are still relatively simple when compared to platforms such as JAVA or .NET. The capacity of the platform to scale up to such platforms must thus be evaluated. Finally, it will be important to apply our choices in an appropriate case study that starts from the security policy specification to the certificate verification (proof of concept).

References

- [1] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. *Certificate translation for optimizing compilers*. In Proceedings of the 13th International Static Analysis Symposium. LNCS, Springer-Verlag, 2006.
- [2] G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa, *Tool-Assisted Specification and Verification of Typed Low-Level Languages*. Journal of Automated Reasoning, Jan 2006.
- [3] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll, *An overview of JML tools and applications*, International Journal on Software Tools for Technology Transfer (STTT) **7** (2005), no. 3, 212–232.
- [4] J.-C. Filliâtre, *Resources for the compilation course - the virtual machine*, from the author’s webpage.
- [5] J.-C. Filliâtre, *Why: a multi-language multi-prover verification tool*, Research Report 1366, LRI, Université Paris Sud, March 2003.
- [6] J.-C. Filliâtre and Claude Marché, *Multi-Prover Verification of C Programs*, Sixth International Conference on Formal Engineering Methods (ICFEM) (Seattle), Lecture Notes in Computer Science, vol. 3308, Springer-Verlag, November 2004, pp. 15–29.
- [7] G. C. Necula and P. Lee, *Efficient representation and validation of logical proofs*, Proceedings of the 13th Annual Symposium on Logic in Computer Science. IEEE Computer Society Press, 1998, pp. 93–104.