Advances in Programming Languages

Certifying correctness

David Aspinall

School of Informatics
The University of Edinburgh

Thursday 28 January 2010
Semester 2 Week 3

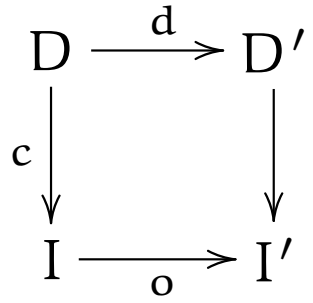
Topic: Some Formal Verification

This is the final lecture of four lectures about some techniques and tools for formal verification, specifically:

- Hoare logic
- JML: The Java Modeling Language
- ESC/Java2: The Extended Static Checker for Java
- Certifying correctness: approaches and examples

Outline

Verification at Different Levels



checking design (or specification)

- check that D satisfies some property

checking implementation

- check that I has some behaviour

checking translations

- check that *refinement* d preserves properties
- check that *compilation* c preserves properties
- check that *optimisation* o preserves behaviours

Certifying Correctness

> **Certify**. *trans*. To make (a thing) certain; to guarantee as certain, attest in an authoritative manner; to give certain information of.

Various mechanisms are used to provide guarantees of checks performed to show software correctness or suitability.

- informal argument written in English
- check-list of manually measured/assessed criteria
- set of executable tests that are checked automatically
- transcript of input and output to a verification system
- a signature of an authority, analogue or digital
- digital evidence, checked electronically

Certifying Correctness

> **Certify**. *trans*. To make (a thing) certain; to guarantee as certain, attest in an authoritative manner; to give certain information of.

Various mechanisms are used to provide guarantees of checks performed to show software correctness or suitability.

- informal argument written in English
- check-list of manually measured/assessed criteria
- set of executable tests that are checked automatically
- transcript of input and output to a verification system
- a signature of an authority, analogue or digital
- digital evidence, checked electronically

Outline

Code signing: worth the bits?

- Currently: we trust code based on authentication of its source.
  I'll trust updates to IE only if they're signed by Microsoft.

- Code signing is better than trusting unauthenticated code: digital version of "shrink-wrapping". But this trust is fallible:
  - Microsoft's signing scheme may be compromised (this has actually happened, by a infamous social engineering attack on Verisign),
  - More seriously, the code might not be secure anyway, if Microsoft fails to program securely, or infection/corruption before signing (also happened: MS accidently distributed Nimda virus with VS .NET!)
- The problem is that we delegate trust to somebody else rather than examining the code for ourselves.
- Could we instead examine the code ourselves to *prove* that it is secure?
  - that seems like hard work, proving all those VCs...
  - but if someone gave us the proofs, we *can* efficiently check them!

Proof-carrying Code

- Ideally, we certify code not to its origin, but with a **self-evident guarantee of security**, to capture exactly what we want.
- The code is packaged together with the guarantee and shipped to the *code consumer* (client).
- The consumer checks:

Proof-carrying Code

- Ideally, we certify code not to its origin, but with a **self-evident guarantee of security**, to capture exactly what we want.
- The code is packaged together with the guarantee and shipped to the *code consumer* (client).
- The consumer checks:
  1. the guarantee is correct
  2. the guarantee ensures the local security property desired
  3. the guarantee matches the code

  If so, the code is safe to execute.

Ex: what can go wrong?

Proof-carrying Code

- Ideally, we certify code not to its origin, but with a **self-evident guarantee of security**, to capture exactly what we want.
- The code is packaged together with the guarantee and shipped to the *code consumer* (client).
- The consumer checks:
  1. the guarantee is correct
  2. the guarantee ensures the local security property desired
  3. the guarantee matches the code

  If so, the code is safe to execute. Ex: what can go wrong?
- This is the subject of research into **proof-carrying code** (PCC) and, more generally, **evidence-based security**.
- A form of evidence-based security (aka "lightweight PCC") is now used in Java: the *stack maps* used in JVML since Java 1.6 (see JSR 202).

Proof-carrying Code: mechanism

- Basic idea: give a **mechanized proof** that security properties are met. The compiler and/or programmer adds annotations to the code to express security-related invariants. These annotations become the **proof certificate** that the code is safe, and can be efficiently checked.

- In practice, the PCC protocol may allow for some negotiation to set security policy.

- It might also allow for the combination of cryptographic and proof certificates.

- Theoretical work of the LFCS institute in Informatics, Edinburgh, dating from 80s–90s is being applied today in PCC.
  Specifically, *Logical Frameworks* are used to explicitly represent proofs, and *Deliverables* are the package of a program plus proof.

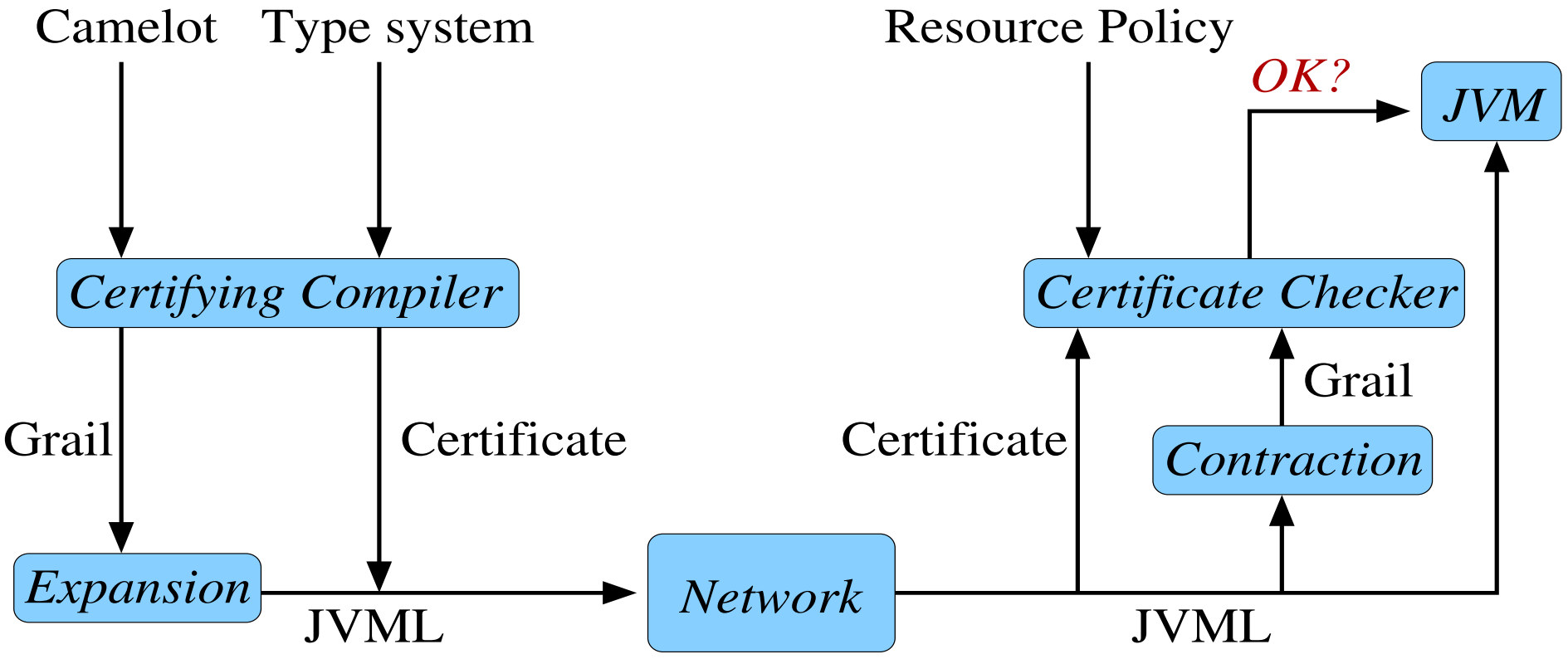
PCC Example: MRG – Mobile Resource Guarantees

- Write programs in a custom high-level language **Camelot**, a functional language with an OCaml-like syntax.
- Camelot is compiled into **Grail**, a functional intermediate code, which is isomorphic to a subset of JVML.
- Use an abstract **cost model** for the JVM which counts instructions and measures stack and heap sizes.
- Costs are calculated using a **annotated operational semantics** for Grail, reflecting the expansion into JVML.
- **Grail Logic** is a Hoare-like program logic which can express resource assertions about the operational semantics.

PCC Example: MRG – Mobile Resource Guarantees

- Write programs in a custom high-level language **Camelot**, a functional language with an OCaml-like syntax.
- Camelot is compiled into **Grail**, a functional intermediate code, which is isomorphic to a subset of JVML.
- Use an abstract **cost model** for the JVM which counts instructions and measures stack and heap sizes.
- Costs are calculated using a **annotated operational semantics** for Grail, reflecting the expansion into JVML.
- **Grail Logic** is a Hoare-like program logic which can express resource assertions about the operational semantics.
- Camelot has a **resource type inference system**, which is used to produce proofs in a **logic of derived assertions**.
- The annotated semantics, logics, and meta-theorems have all been formalised in **Isabelle**, and Isabelle proof scripts are used as a proof transmission format.

Architecture of MRG



Camelot

- Example program: insertion sort:

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

- The notation `@_` indicates a destructive pattern match
- Whole program compilation where each Camelot function yields one JVM method
- Compilation includes an explicit memory manager (freelist)

Program analysis, certification and proof checking

```
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)
```

- Memory consumption inferred from program annotations using a type system; usage expressed relative to size of input.
- Here: `ins` consumes one memory cell, independent from actual input, `sort` does not consume any memory (in-place)
- PCC certificate: the result of type inference is encoded in a program logic for the compiled Grail code
- Certificate bundled with Java class file for transmission
- JVM at consumer side uses modified class loader that checks certificate in Isabelle before executing the program.

MRG demos

See:

- MRG home page at http://groups.inf.ed.ac.uk/mrg/
- MRG demo at http://projects.tcs.ifi.lmu.de/mrg/pcc4/index.php

usually working, sometimes goes down

PCC Example: Mobius — Mobility, Ubiquity and Security

- An EU project 2004-2009 with 16 partners in 10 countries.
- Started from a shared concept of proof-carrying code
- Extended to gather in a range of types of digital evidence that guarantee program behaviour.

  proof based The certificate contains a proof which refers to bytecode behaviour in a bytecode logic. A proof-checker checks these.

  type based The certificate contains typing annotations for a specialised type system which extends Java typing and guarantees a safety invariant. A type-checker checks the annotations. Separately and offline, the type system is proved correct.

  abstract-interpretation based The certificate contains solutions for an program analysis problem based on constraint solving. The solution is checked to satisfy the constraints.

Mobius Demos

Video on YouTube

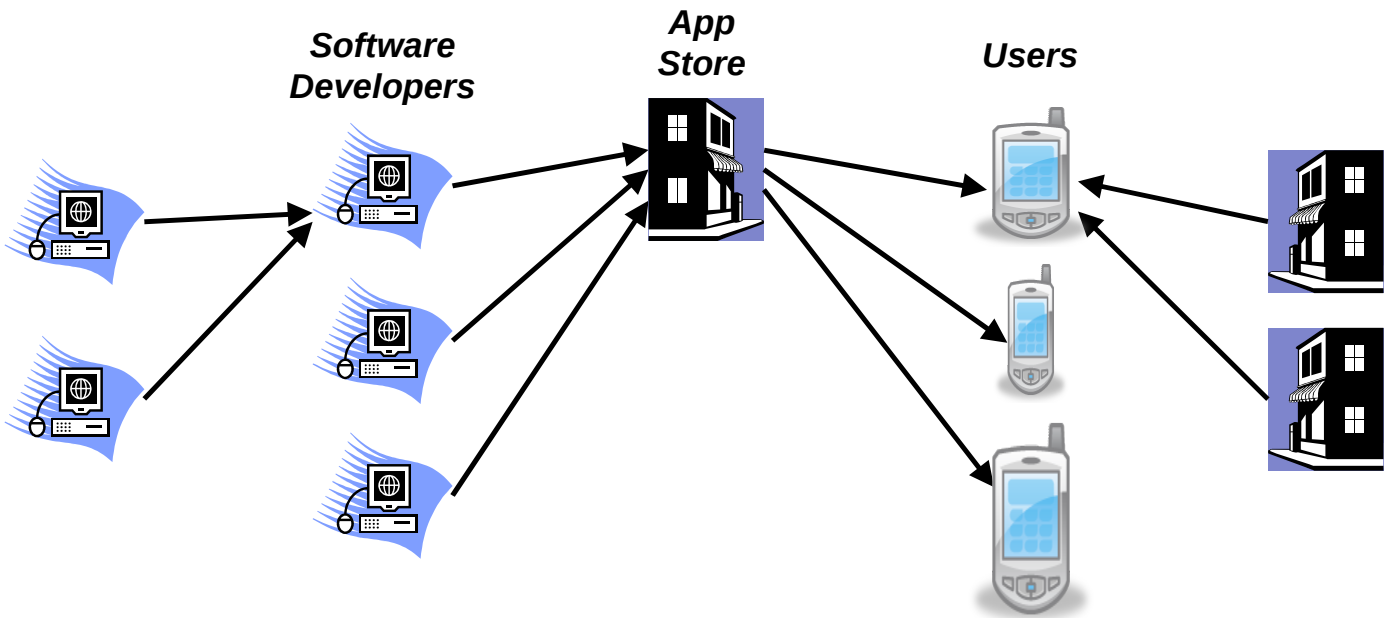
- Search for "Mobius demo" or go to direct link
  http://www.youtube.com/watch?v=4AYiwo4NQtE

Mobius Quiz (uses simulator)

- See Ian's pages at http://homepages.inf.ed.ac.uk/stark/mq/

currently *partially* working, may be fixed soon

Possible future: Trustworthy Apps



Digital evidence flows around the Trustworthy App Store network architecture:

- From store to user, evidence to satisfy security/resource policy
- From store to developer, stating objective acceptance policies
- From developer to store, providing evidence to meet these

credit: this slide is due to Ian Stark

Outline

Compiler correctness

Compiler correctness is an old example of the translation correctness problem. Modernly, two viewpoints:

theoretical folk

> Compilers are complicated pieces of code.
> We rely on them, but they are buggy.

Compiler correctness

Compiler correctness is an old example of the translation correctness problem. Modernly, two viewpoints:

theoretical folk

> Compilers are complicated pieces of code.
> We rely on them, but they are buggy.
> Like other complicated pieces of code we ought to verify that they do what is intended, and fix them if they don't.

Compiler correctness

Compiler correctness is an old example of the translation correctness problem. Modernly, two viewpoints:

theoretical folk

> Compilers are complicated pieces of code.
> We rely on them, but they are buggy.
> Like other complicated pieces of code we ought to verify that they do what is intended, and fix them if they don't.

compiler folk

> Compilers are *very* complicated pieces of code.
> We rely on them and they are buggy. But that's life.

Compiler correctness

Compiler correctness is an old example of the translation correctness problem. Modernly, two viewpoints:

theoretical folk

Compilers are complicated pieces of code.
We rely on them, but they are buggy.
Like other complicated pieces of code we ought to verify that they do what is intended, and fix them if they don't.

compiler folk

Compilers are *very* complicated pieces of code.
We rely on them and they are buggy. But that's life.
The underlying hardware also has bugs. There haven't been any verified CPU cores since the 1980s. We just hope the bugs are rare, and only use old CPU architectures for controling nuclear power stations.

Piecemeal Verification: Translation Validation

- The idea of **translation validation** is rather than to verify a whole compiler, check that *individual* compilations (or individual optimisation steps) actually performed are correct.
- This relaxes the requirement that the compiler is globally correct. The compiler checks that it has produced the right result each time.
- Some input programs may produce errors or trigger compiler bugs; validations cannot be produced for these programs.

See:

Amir Pnueli, Michael Siegel and Eli Singerman. *Translation Validation*, TACAS '98. http://portal.acm.org/citation.cfm?id=691453

George C. Necula. *Translation validation for an optimizing compiler*, SIGPLAN Notices, 35/5, 2000. http://doi.acm.org/10.1145/358438.349314.

Compcert: Compiler Verification Revisited

- Compcert (developed at Inria in Paris, France) uses **Clight**, a subset of C. Compilation is to a real architecture, **PowerPC**, and with a realistic optimisation level.
- Notion of correctness is formally established:

$$S \text{ safe} \Longrightarrow \forall B, C \Downarrow B \Longrightarrow S \Downarrow B$$

  where safe means "does not go wrong"
- 14 stages through 7 intermediate representations, including register allocation, instruction scheduling, layout of stack frames, etc.
- Formal proofs are carried out in the **Coq** interactive proof assistant
- The compiler itself is coded directly in a pure functional way inside Coq's logic, simplifying reasoning (no Hoare logic is needed)
- The code can be *extracted* to a speedy OCaml program.
- The complete work is available as commented source code at http://compcert.inria.fr/.

Outline

Formal Verification: Overall Summary

Correctness approaches

- **design** checking that the design (or specification) has good properties
- **implementation** checking that the implementation has the right behaviour
- **translation** checking that an implementation is a correct realisation of the design.

Examples

- **program verification**: checking a program meets its specification
- **proof-carrying code**: program verification with electronic evidence
- **translation validation**: checking particular translations are correct
- **compiler verification**: checking that a compiler translates correctly

Example projects and tools: Hoare logic, JML, ESC/Java2, MRG, Mobius, Compcert.

Outline

Reading for next week

Next block of lectures

- The next block of lectures will start out by considering notions of inheritance, subtyping and how they fit together, using Haskell types.

- Before Monday, you should download and read the following paper:

  William R. Cook.
  Interfaces and specifications for the Smalltalk-80 collection classes.
  Proc. OOPSLA 1992.
  http://doi.acm.org/10.1145/141936.141938