Advances in Programming Languages

APL12: Language Augmentations and Correctness

David Aspinall
(some slides by Ian Stark)

School of Informatics
The University of Edinburgh

Friday 5 November 2010
Semester 1 Week 7

http://www.inf.ed.ac.uk/teaching/courses/apl

Topic: Augmented Languages for Correctness

The next three lectures will be about some language-based techniques and tools for improving program quality, specifically:

- Augmentations and Certifying Correctness
- Assertions and Hoare Logic
- Practical tools for Java Correctness

This first lecture sets the scene, introducing the idea of language *augmentations* and describing their use in different forms of correctness checking.
Detailed examples of programs augmented with correctness claims appear in the next two lectures.

Outline

1. Augmented Programming
2. Certifying Correctness
3. Proof-Carrying Code
4. Certified Compilation
5. Summary

Outline

1. Augmented Programming
2. Certifying Correctness
3. Proof-Carrying Code
4. Certified Compilation
5. Summary

Programming Language Features vs Augmentations

A distinction (non-standard terminology):

language features are ways to *determine* behaviour

- exceptions are a way of handling errors

language augmentations are ways to *describe* behaviour

- Java's **`throws`** declaration describes possible errors
- Java's **`@NonNull`** annotation suggests `null` isn't allowed

Language features are realised by language extensions or libraries. Or perhaps by meta-programming or DSLs.

Language augmentations may be realised by language extensions or *annotations*, appearing as a language construct, formal comments, or stored in auxiliary files.

Part of the language or not?

Question: is a static type system an augmentation or part of a programming language?

Part of the language or not?

Question: is a static type system an augmentation or part of a programming language?

An augmentation: static types in many languages do not affect the meaning of programs. They are used to reduce the set of acceptable programs to preclude undesirable behaviours.

For example, static typing usually ensures that *well-typed programs do not go wrong* by adding an integer to a boolean.

Part of the language or not?

Question: is a static type system an augmentation or part of a programming language?

An augmentation: static types in many languages do not affect the meaning of programs. They are used to reduce the set of acceptable programs to preclude undesirable behaviours.

For example, static typing usually ensures that *well-typed programs do not go wrong* by adding an integer to a boolean.

Part of the language: sometimes a static type-system *can* affect the meaning of a program. Then it is properly part of the language, as programs without types may not have meanings.

For example, the *ad-hoc polymorphism* provided by Haskell type classes causes code to be compiled differently for different types.

Augmentations for Correctness, Safety, Security, ...

Key characteristic: augmentations *do not change the meaning of the program*. But they may be used to *express properties* of the program. These properties may be checked by a compiler or other tools.

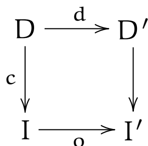
- error behaviour
  - e.g., (runtime) exception freedom: no `NullPointerException`
- resource usage
  - e.g., a method requires 50kb of heap space
  - ... or takes time proportional to the length of its input list
  - ... spawns at most 4 threads
- **functional correctness**
  - e.g., the method `Math.log(x)` computes the natural logarithm $\ln(\underline{x})$

Functional correctness is generally the most ambitious; the aim of **formal verification**. Although by Rice's theorem, the properties are most likely equivalent.

Outline

1. Augmented Programming
2. Certifying Correctness
3. Proof-Carrying Code
4. Certified Compilation
5. Summary

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/behaviours
- 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

1. Augmented Programming
2. Certifying Correctness
3. Proof-Carrying Code
4. Certified Compilation
5. Summary

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. Instead, could we examine the code ourselves to *prove* that it is secure?
  - that seems like hard work...
  - but if someone gives us the proofs, we can easily 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 ("lightweight PCC") is 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.

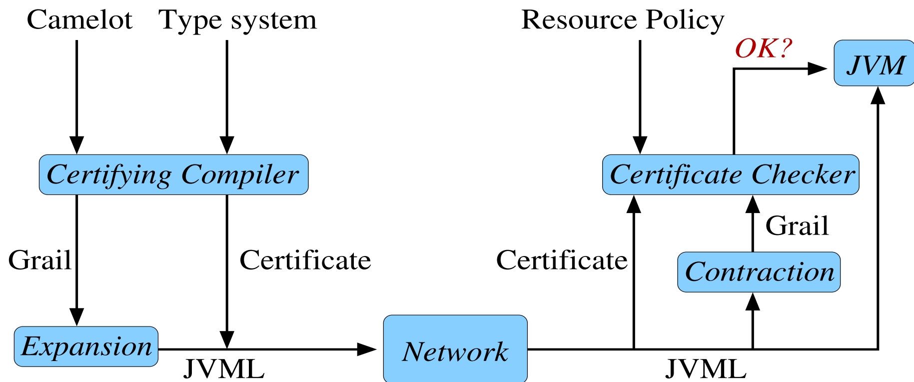
- It might also allow for the combination of cryptographic and proof certificates.

- 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.
- An abstract **cost model** for the JVM counts instructions and measures stack and heap sizes.
- Costs are calculated using a **annotated operational semantics** for Grail, reflecting the expansion into JVML.
- Camelot has a **resource type inference system**, which is used to produce formal proofs automatically for a form of Hoare Logic.
- 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. This affects the behaviour, but augmentations for *cost accounting* are available to describe the resource usage and give hints to the resource analysis.
- Compilation includes an explicit memory manager whose space usage is inferred using a type system and program annotations.
- 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.

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

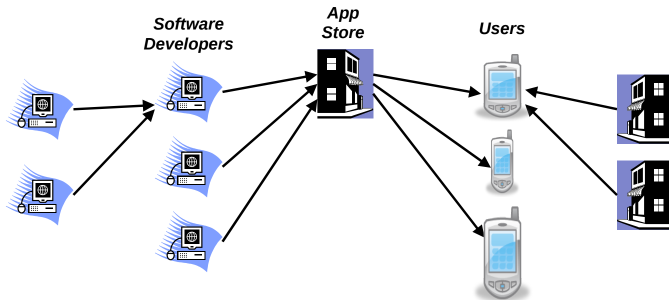
- Search for "Mobius demo" or go to direct link http://www.youtube.com/watch?v=4AYiwo4NQtE

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

1 Augmented Programming

2 Certifying Correctness

3 Proof-Carrying Code

4 Certified Compilation

5 Summary

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:

  > If S is *safe*, then for all behaviours B, if a compiled program C has behaviour B then the source program S has behaviour B too.

  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

1 Augmented Programming
2 Certifying Correctness
3 Proof-Carrying Code
4 Certified Compilation
5 Summary

Summary: Augmentation and Correctness

Language augmentations

Augmentations are *formal* extensions or additions to programs, which express properties about programs but do not affect their behaviours. They may be exploited by the language compiler and/or auxiliary tools.

Correctness approaches

- **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 research projects and tools: MRG, Mobius, Compcert.

Reading before next week

Foundations of formal verification: Hoare logic

- You should review predicate calculus and logic notation

$$P \land Q \qquad P \lor Q \qquad P \to Q \qquad \neg P$$

$$\forall x. P \qquad \exists x. P$$

Practical languages for verification: JML and ESC/Java 2

- First you should understand the dynamic analogue, runtime checking with *assertions*. Read the Java assertions tutorial at:
  http:
  //download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html