Resource Guarantees and PCC

50 ways* to say it with a proof

Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics
The University of Edinburgh

Proof Carrying Code workshop
Friday 11 August 2006

*Note: Contents may vary

Typed Java

We know that:

- Well-typed programs don’t go wrong
- All Java programs are strictly typed

So we deduce that:

- No Java program will go wrong

Even better, Java is typed both at source and bytecode level, so we can conclude this twice over.

Sadly, life is not so simple. Typing in Java and Java bytecode is a good thing, but not the end of the story; and PCC is one means to add to its effectiveness.

Talk Overview

Mobile Resource Guarantees

- PCC for guaranteed bounds on time and heap space
- Certified code runs on standard Java virtual machine
- Certifying compiler infers resource types for a high-level ML-like source language
- Guarantees are proofs in resource-aware bytecode logic

Varieties of Proof-Carrying Code

- Configurations for code producers and consumers
- Resource policy language
- Validation of compiler optimisations
- Probabilistically checkable proofs
- e-Science and the Grid

Mobile Resource Guarantees (MRG)

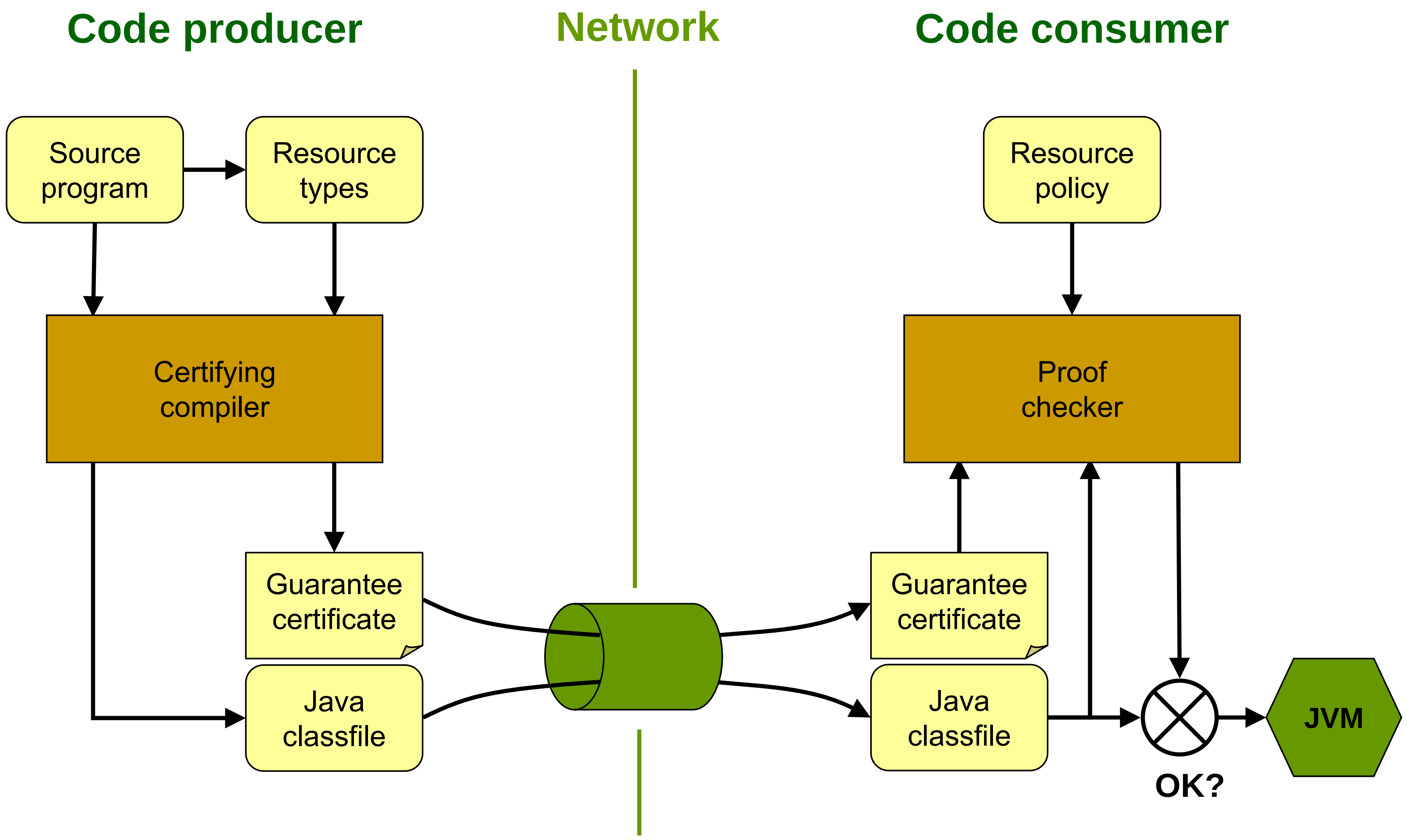
MRG: 3-year research collaboration between Edinburgh and Munich, funded by the European Commission as a “Future and Emerging Technology” in global computing.

The aim was to implement a proof-carrying code framework for Java bytecode providing guarantees of resource usage.

Java bytecode: Standardised and portable virtual machine.

Resource guarantees: Practically useful and more tractable than verifying full correctness.

MRG Framework



Camelot

[Mackenzie/Wolverson '04]

```
type intlist = !Nil | Cons of int * intlist

let rev l acc = match l with
    Nil => acc
  | Cons(h,t)@d => rev t (Cons(h,acc)@d)

let reverse l = rev l Nil
```

ML-like source language with:

- Explicit heap cell manipulation `Cons(h,t)@d`
- Freelist annotations `(x,xs)@_`
- Hooks to Java objects and libraries `obj#meth x y`

Compiles to Java bytecode, executes on standard JVM.

Space Types

[Hofmann/Jost POPL ‘03]

```
let insert e l =
  match l with [] -> e::[]
             | (h::t)@_ -> if h >= e then e::(h::t)
                                     else h::(insert e t)
let sort l =
  match l with [] -> [] | h::t -> insert h (sort t)
```

Type system to describe heap cell usage:

```
insert: 1, list(0) -> list(0), 0
sort  : 0, list(1) -> list(0), 0
```

- Input and output heap sizes related to argument and result data sizes respectively: gives composability.
- Includes user-defined datatypes.
- Types inferred via a separate linear constraint solver.

Grail

[Beringer/Mackenzie/S. ‘03]

```
fun loop(int n, int a) =
    let val a = mul a n          // Multiply into
        val n = sub n 1          // accumulator,
    in                           // decrement and
        test(n,a)                // test again.
    end
```

Highly structured form of Java bytecode.

- Functional view, target for Camelot compiler.
- Imperative view as static single assigment.
- Views are equivalent, including resource usage.
- Binary format is executable `.class` files, with tools to interconvert with functional presentation.

Tight constraints on control flow and stack use greatly simplify reasoning about bytecode.

Bytecode Logic

[Aspinall/Beringer/Hofmann /Loidl/Momigliano ’04]

Assertions: $G \rhd e : P$

Predicates: $P : \mathit{Env} \times \mathit{Heap} \times \mathit{Heap} \times \mathit{Value} \times \mathit{Resource} \to \mathit{Bool}$

Validity: $\models e{:}P$ iff $\forall E,h,h',v,e \,.\, E \vdash h,e \Downarrow h',v,r \;\Rightarrow P(E,h,h',v,r)$

Logic of assertions about Grail code fragments

- Syntax-directed derivation rules for $G \rhd e : P$
- Sound and complete for Grail operational semantics
- Isabelle implementation (shallow embedding)
- Resource aware: heap size, stack depth, instructions,…
- Applies to complete recursive object-oriented programs
- Partial correctness — separate logic for termination

Resource Algebras

[Aspinall/Beringer/ Momigliano '06]

Grail operational semantics and bytecode logic refer to an element $r \in \mathit{Resource}$ from an arbitrary resource algebra:

$$E \vdash h, e \Downarrow h', v, r \qquad G \rhd e : P \quad : \ldots \times \mathit{Resource} \rightarrow \mathit{Bool}$$

Resource algebra = Partially ordered monoid $(\mathcal{R}, 0, +, \leq)$ with constants $\mathcal{R}^{\mathit{int}}, \mathcal{R}^{\mathit{getf}}, \mathcal{R}^{\mathit{if}}, \ldots$ for all Grail operations

Current implementation measures heap space, instructions, jumps, method invocations and maximum call depth.

Other possibilities: stack size, specific method invocation, safety flags, system calls, even a complete instruction trace.

Automated Certification

[Momigliano/Shkaravska/ Beringer/Hofmann '04]

Encode Camelot resource typing

$$n, \Gamma \vdash e : A, m$$

as a bytecode logic expression

$$G \rhd e : [\![U,n,\Gamma \blacktriangleright A,m]\!]$$

(U is a set of variables for checking linearity)

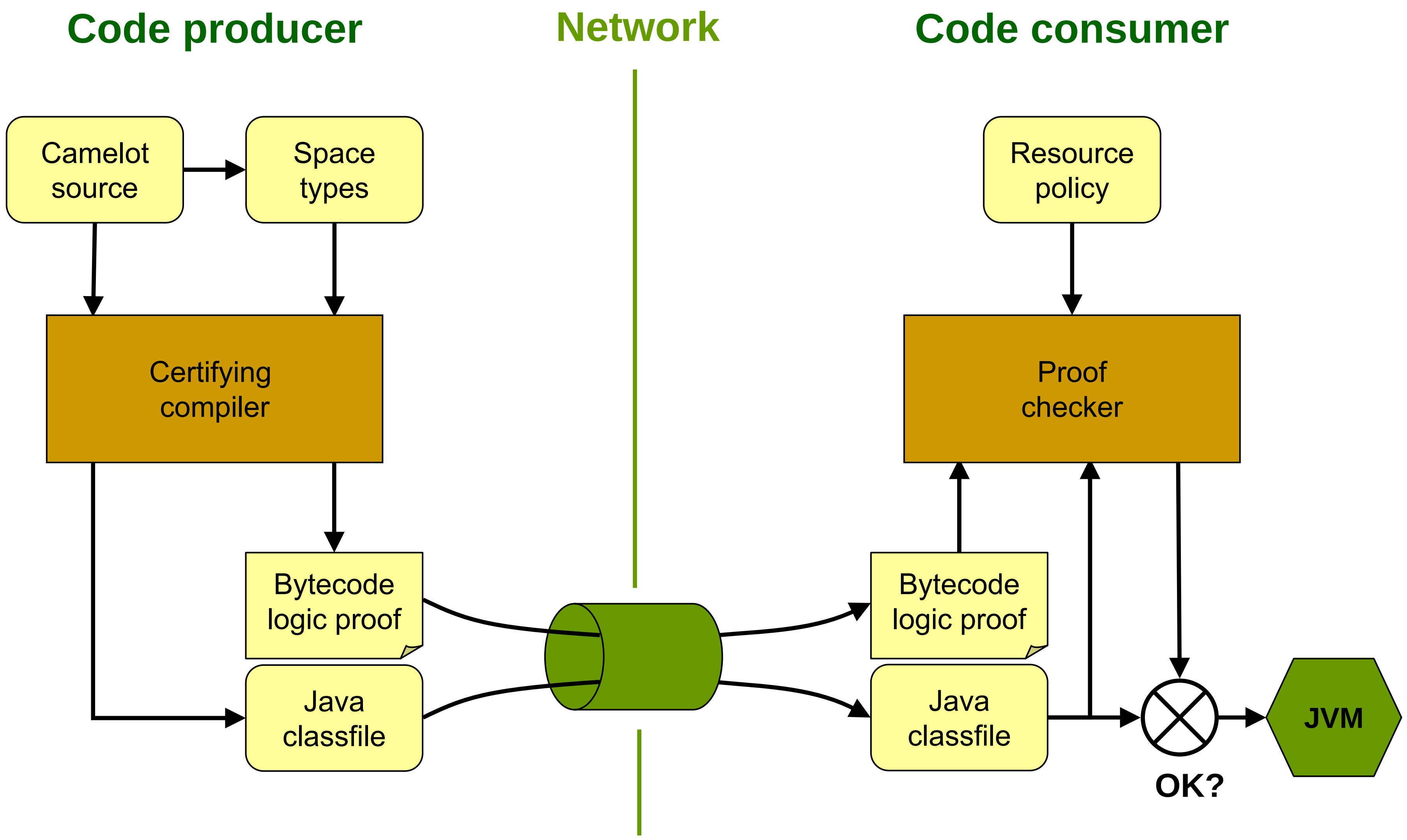
Form $[\![U,n,\Gamma \blacktriangleright A,m]\!]$ expands to a predicate describing a freelist on the heap of the necessary size.

These predicates form a derived logic: all typing rules give valid deductions in the derived logic.

So resource typings compile to resource logic proofs – in fact, to tactics that execute these proofs in Isabelle.

It is these tactics that are parcelled up with bytecode.

MRG Framework



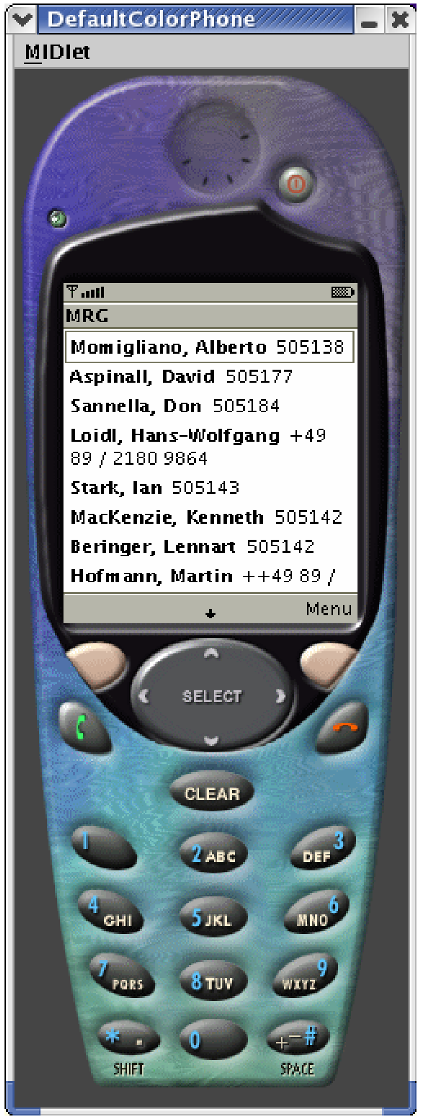
Demonstration

Departmental phonebook application

Buzzword profile:

- J2ME – Java 2 Micro Edition
- CLDC – Connected Limited Device Configuration
- MIDP – Mobile Interactive Device Profile

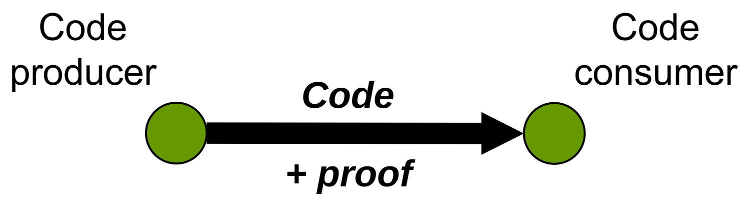
i.e. it runs on a cellphone, smartphone or PDA



Momigliano, Alberto 505138
Aspinall, David 505177
Sannella, Don 505184
Loidl, Hans-Wolfgang +49 89 / 2180 9864
Stark, Ian 505143
MacKenzie, Kenneth 505142
Beringer, Lennart 505142
Hofmann, Martin ++49 89 /

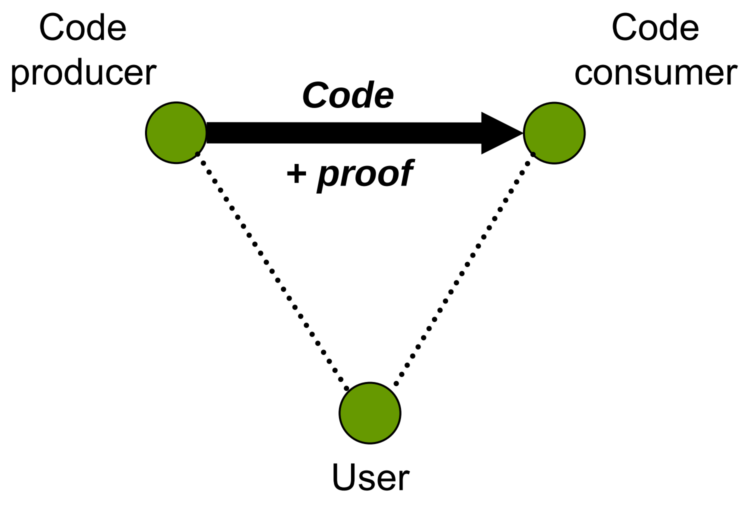
PCC configurations

The simplest arrangement for proof-carrying code is this:



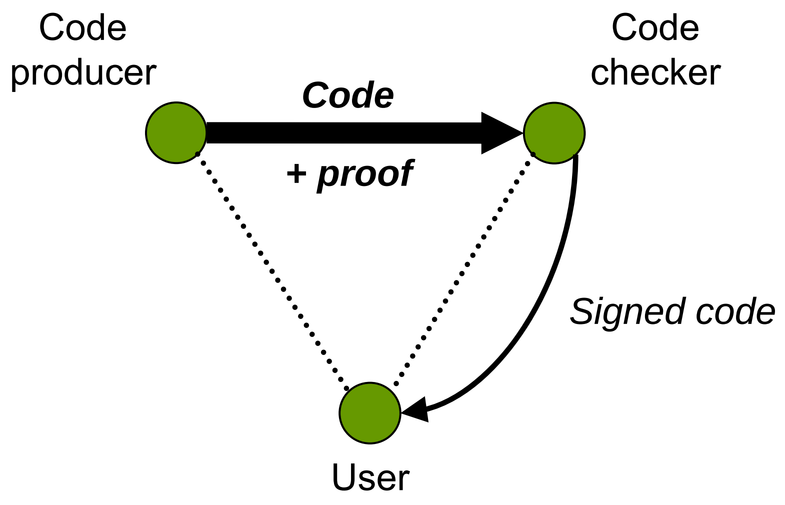
PCC configurations

In the MRG demonstration, there was a third party controlling the interaction:



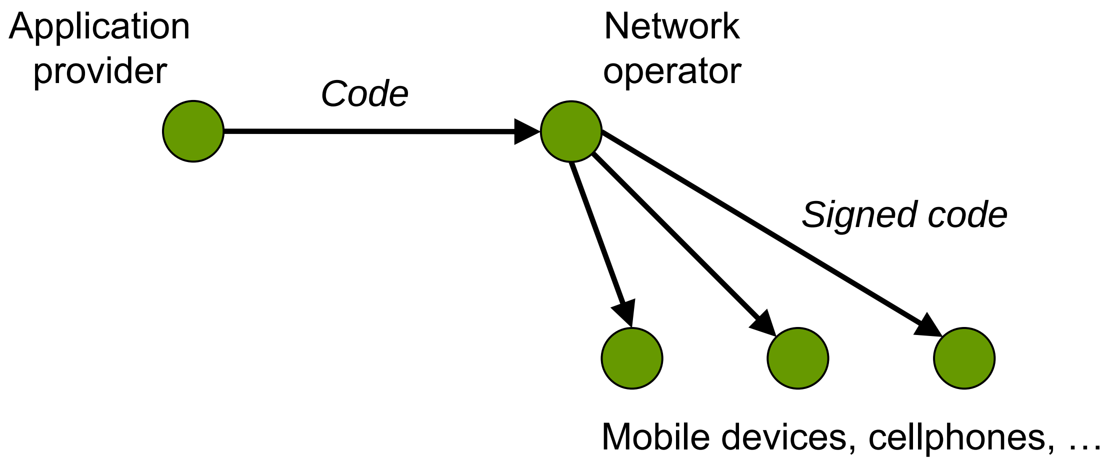
PCC configurations

Once code has been independently checked, it might be signed and sent on to the user:



PCC configurations

Mobius – Mobility, Ubiquity and Security: EC project on PCC for Java on small devices, such as mobile phone *midlets*.

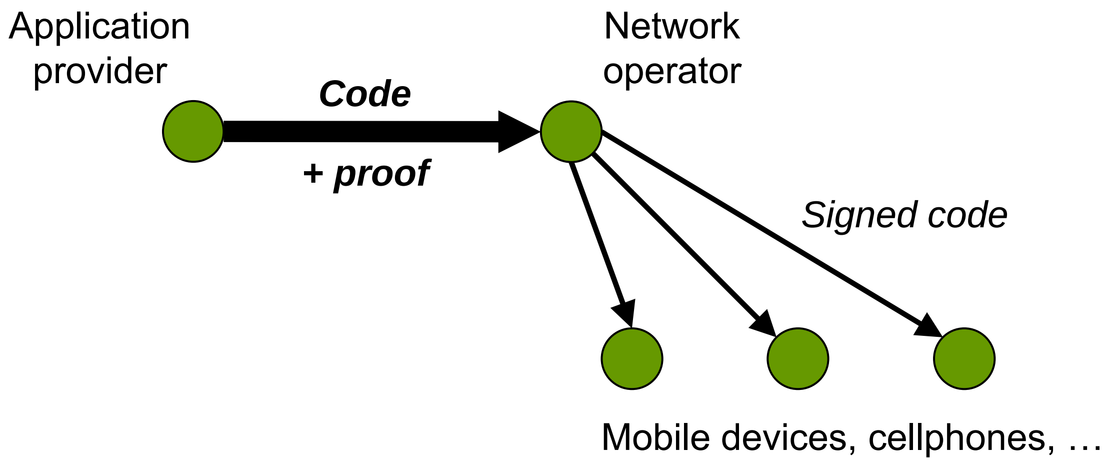


Resource use (memory, cpu, network) is a significant issue.

Network operators currently test and sign code variations for each different kind of handset.

PCC configurations

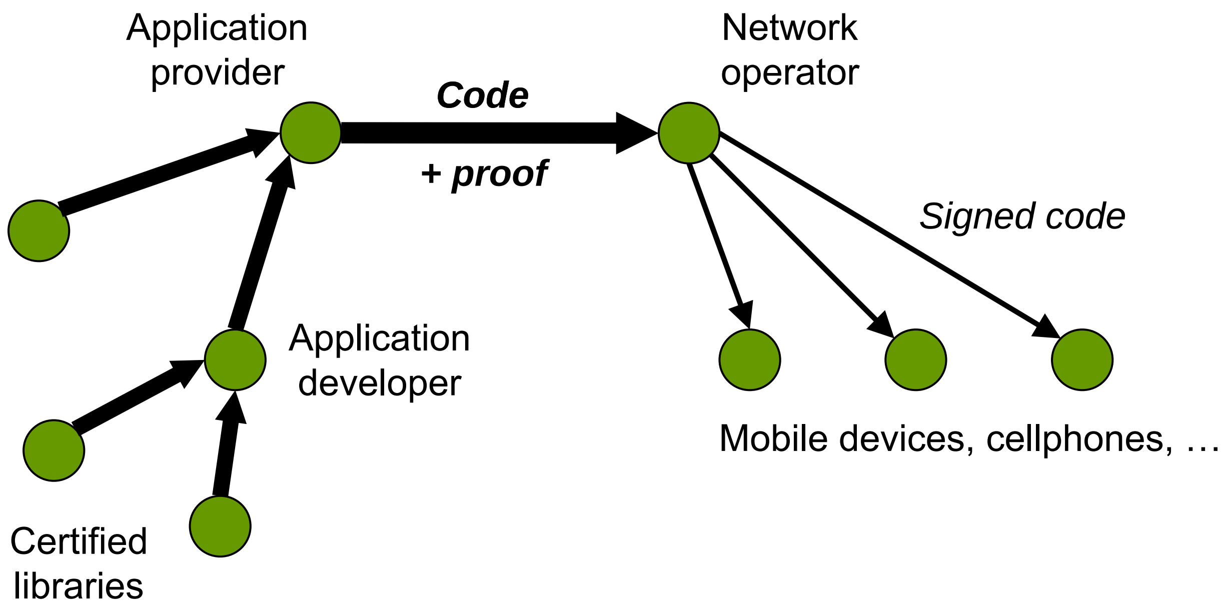
Application providers and network operators already have a trust relationship, which PCC can strengthen:



Private source code: PCC supports validation without source release, which is essential in practice.

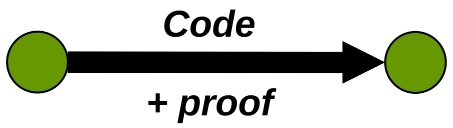
PCC configurations

The chain may be longer, with developers who in turn use libraries, and all parties seeking guarantees of safety:

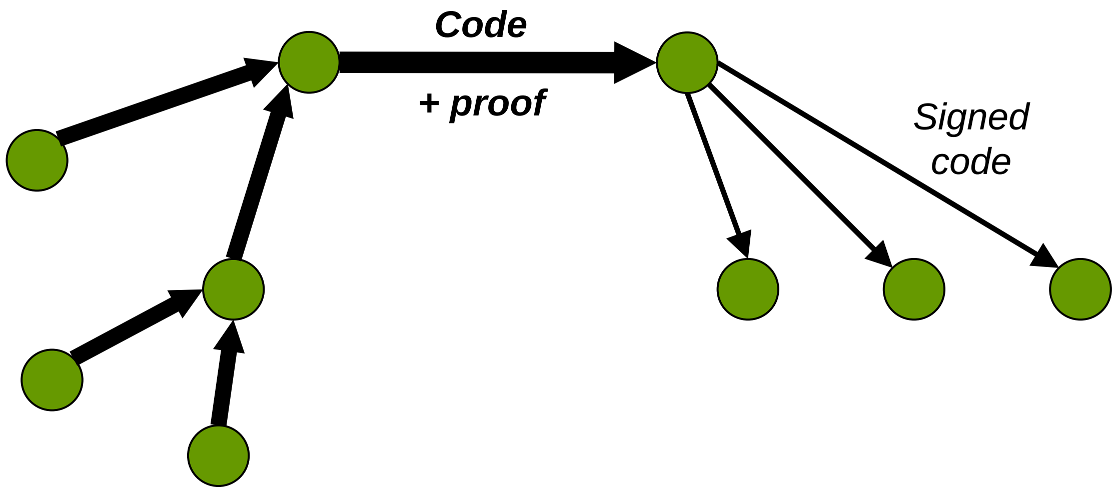


PCC configurations

In general we can distinguish between:



Retail PCC



Wholesale PCC

Policies

[Aspinall/Mackenzie '06]

Using proofs of resource safety demands some flexibility:

- Programs are not simply correct/incorrect: this depends on the resources available
- Different target devices have different resource profiles

We address this by identifying

- A guaranteed resource policy R of actual code behaviour
- A target resource policy T of what the client will accept

The certificate checker must confirm

1. $\triangleright P : R$ The program does indeed meet its guarantee
2. $R \preccurlyeq T$ The guarantee ensures the target

We have a language for policies to suit both (1) and (2).

Sample Resource Policies

Implementation builds on the Java security policy framework:

For positive integer inputs *m* and *n*, the method call `calc(m,n)` requires at most 2*n* bytes of heap space

```
permission SpaceGuarantee "calc(int m,int n)" "2*n"
```

For all positive integer inputs $m < 3$ and $n < 4$, executing `calc(m,n)` must take no more than 500 instructions

```
permission ClockTarget "calc(int m,int n)" "500,m<=3,n<=4"
```

For positive integer input *n*, method `C.k(n)` takes no more than 4*n*+3*d* instructions, where `D.rand(n)` takes *d* instructions.

```
permission ClockGuarantee "C.k(int n)"
                 "4*n + 3*(ClockGuarantee D.rand(n))"
```

Policies may use +, *, ^, log, min, max (all non-decreasing)

Using Resource Policies

The prototype implementation builds on the existing Java security framework:

- Permissions, policies, and policy configuration files;
- Java ‘agents’ offer a route to enforce resource policies at class loading time.

Policy comparison is very fast, thus:

- Code might carry multiple guarantees for different targets or different kinds of resource; and
- Code consumers could mix and match (dependent) guarantees from separate certified components to meet a given target policy.

Optimisation Validation [Aspinall/Beringer/Momigliano '06]

Classic translation validation proves that compiler transformations preserve the meaning of programs.

By analogy, optimisation validation is proving that ‘optimising’ transformations do improve things.

The MRG bytecode logic and resource algebras give a suitable framework to carry out optimisation validation.

- Examples like constant folding, dead code elimination, some loop unrolling.
- Independent of any validation of functional correctness.
- Static resource algebras for static costs like code size.

Note that all this is distinct from the issue in PCC of preserving proofs about programs through compiler optimisations.

Tactic-carrying Code

For proof validation, there is in general a trade-off between size of data sent and computation required by the verifier.

In MRG we send Isabelle tactics, which guide (re)generation of proof by the verifier. (15 lines of tactic ~ 34k lines of proof)

Extreme example: There exists a proof of size *k*; find it. (Checking decidable but slow)

Classically faster checking needs more data; with a lower bound that the verifier must at least traverse the whole proof.

This is not however the whole story...

Probabilistically Checkable Proofs

How to check a proof, probably:

1. Producer converts proof p to transparent form p'.
2. Verifier inspects some (randomly chosen) bits of p'.
3. If p valid, check succeeds; if not, fixed chance of failure.
4. Rinse and repeat to reach desired confidence.

Sizes: $|p'| = O(|p| \log |p|)$; fraction checked is logarithmic; but can be fixed, as small as 3 bits.

Refinement: Don't send p', instead use data commitment protocol at cost $\log|p'|$ per bit inspected.

Good news: (asymptotically) less computation for verifier *and* less data sent.

Bad news: Large constants, lots of work for the producer.

Probabilistically Checkable Proofs

[Klin '05]

Standard LFKN protocol applies to proving a claim that:

$$\sum_{x_1 \in H} \dots \sum_{x_n \in H} f(x_1, \dots, x_n) = a$$

for a particular polynomial *f*() and value *a*.

We want to prove a claim like:

There is a derivation of resource typing $m \vdash e : n$.

The answer is to arithmetise: from the derivation graph structure construct a low-degree polynomial *f* in which is zero iff such a graph exists, i.e. iff there is a well-formed resource typing derivation.

PCP for PCC

[Klin '05]

Klin implemented probabilistic proof-carrying code for a tiny language with types expressing heap space use.

PPCC was feasible but computationally expensive: >30min to build the 10G transparent proof for a 256-node derivation (and this must be done for each interactive protocol run).

Viable for PCC if:

- Substantial asymmetry between producer and consumer
- Computation + storage sufficiently faster/cheaper than communication
- Proofs are large (much larger than code)

So, just wait long enough.

Grids and e-Science

There are (at least) two varieties of “Grid” with opportunities for proof-carrying code:

- Computational grids: remote access to banks of standard machines. Booking and job control.

  (LHC grid, EGEE)

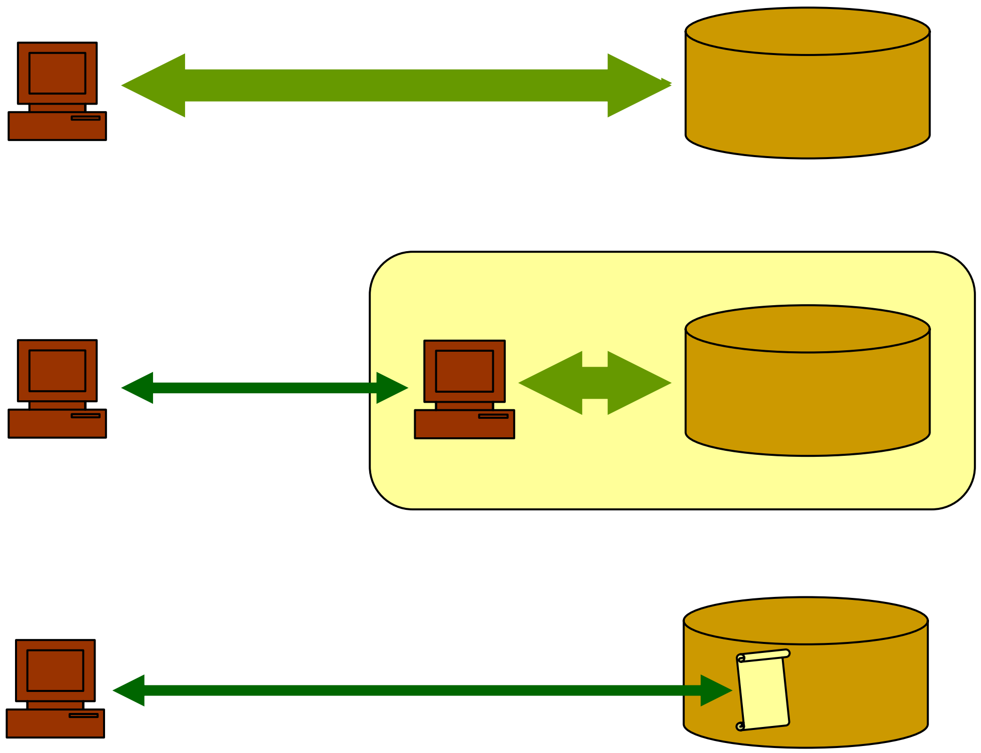
- Data grids: very large databases open to many remote users running complex code.

  (AstroGrid, Virtual Mouse Atlas)

ReQueST: Resource Quantification for e-Science Technologies

Project at Edinburgh to work on PCC for the grid.

Running Code on a Data Grid



Running code closer to the data is faster, but requires more trust.

Some computations may only be feasible as database stored procedures, called from within queries themselves.

Grid PCC

Example datagrids:

Astronomy

- UK AstroGrid virtual observatory; Sloan Digital Sky Survey (SDSS)
- Novel indexing methods (e.g. trixels) and libraries (algebras of regions) driven by spherical geometry

Biology

- Edinburgh Mouse Atlas (emap). Has well established relationships with remote users (not PCC but CCP), seeking to move this to web services

SQL Stored Procedures

Most databases now support some form of stored procedures:

- Originally just C++, but now increasing support of Java
- Conventionally restricted to database builders, for security, but would be useful for all users
- Query optimizers already use assertions about stored procedures (e.g. commutativity, cost) claimed by the programmer

PCC certification could make stored procedures more flexible and widely available to datagrid users.

Review

MRG framework

- High level language; space type inference
- Grail: analysis-friendly Java bytecode
- Resource-sensitive bytecode logic; resource algebras
- Proof tactic scripts for certificates

Variations on resource PCC

- Retail vs. Wholesale PCC
- Resource Policies
- Optimisation Validation
- Probabilistically Checkable Proofs
- Grid PCC

Conclusion

Even with a restricted domain like proofs of resource use, we discover that PCC can be applied at many points in software development and delivery, and in many ways, to many ends.

Each of these adds assurance that programs will not go wrong in some way, which leads to the suggestion that:

Proofs are the new types

Projects

MRG: Mobile Resource Guarantees

Edinburgh, LMU Munich
European Commission project IST-2001-33149

Mobius: Mobility, Ubiquity and Security

INRIA, Zurich, Nijmegen, Munich, Edinburgh, Tallinn, Chalmers, London, Dublin, Warsaw, Madrid, Aachen; France Telecom, Trusted Logic, SAP
European Commission project IST-15905

ReQueST: Resource Quantification for e-Science Technologies

Edinburgh
UK EPSRC e-Science Programme

References

Aspinall, Gilmore, Hofmann, Sannella and Stark. **Mobile Resource Guarantees for Smart Devices**. In *Proc. CASSIS 2004,* LNCS 3362. Springer-Verlag.

MacKenzie and Wolverson. **Camelot and Grail: resource-aware functional programming on the JVM**. In *Trends in Functional Programing*. Intellect, 2004

Hofmann and Jost. **Static prediction of heap space usage for first-order functional programs**. In *Proc. POPL 2003.* ACM Press.

Beringer, MacKenzie and Stark. **Grail: a Functional Form for Imperative Mobile Code**. In *Foundations of Global Computing 2003,* ENTCS 85.1. Elsevier.

Klin. **Probabilistically Checkable Proofs for Mobile Resource Guarantees**. 2005.

Aspinall, Beringer, Hofmann, Loidl and Momigliano. **A Program Logic for Resource Verification**. In *Proc. TPHOLs 2004,* LNCS 3223. Springer-Verlag.

Beringer, Hofmann, Momigliano, Shkaravska. **Automatic Certification of Heap Consumption**. In *Proc. LPAR 2004,* LNCS 3452. Springer-Verlag.

Aspinall, MacKenzie. **Mobile Resource Guarantees and Policies**. In *Proc. CASSIS 2005*, LNCS 3956. Springer-Verlag.

Aspinall, Beringer, Momigliano. **Optimisation Validation**. In *Proc. COCV 2006*, LNCS. Springer-Verlag.

Atkey, MacKenzie *et al.* **ReQueST: Resource Quantification for e-Science Technologies**. In *Proc. PCC 2006.*