Mobile Resource Guarantees

Ian Stark

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

David Aspinall, Stephen Gilmore, Don Sannella,
*Kenneth MacKenzie, *Lennart Beringer, Michal Konečný
LMU Munich: Martin Hofmann, Hans-Wolfgang Loidl, Olha Shkaravska

OUCL Friday 14 March 2003

Mobile Resource Guarantees

MRG is a joint Edinburgh / Munich project funded for 2002–2005 by the European initiative in *Global Computation.*

Our aim is to develop an infrastructure that endows mobile code with independently verifiable certificates describing resource requirements.

We plan to do this by mapping resource types for high-level programs into proof-carrying bytecode that runs on the Java virtual machine.

I'll talk about progress over the first year, and in particular some properties of our *GRAIL* intermediate language.

(LFPL + PCC / JVM)

Global Computation

Programs that travel over networks between computers and other devices, running in different places at different times. For example:

- Mobile phones downloading new software for extra features
- Smartcards that host multiple functions
- Desktop applications exchanging code with web services

Some common features:

- Users expect continuous upgrading, customization and flexibility
- Self-service of mobile code from multiple providers
- Heterogenous clients with irregular resource limitations

Authentication for mobile code

Java

- Originally, Java used a *sandbox* model, where all remote code was wholly untrusted.
- In version 1.2 this moved to more finely grained *security policies* managed through cryptographic signatures on code.

Windows

- Microsoft *Authenticode* also uses cryptographically signed code.
- User can distinguish code from different providers.
- Very widely used – more or less compulsory in XP for drivers.

Useful as these are, they say nothing about the code itself, only its supplier.

Trust me

Security Warning
Do you want to install and run "Provides Files to Add Active Debugging to Hosts and Engines" signed on 7/27/2000 10:29 AM and distributed by:
Microsoft Corporation
Publisher authenticity verified by VeriSign Commercial Software Publishers CA
Caution: Microsoft Corporation asserts that this content is safe. You should only install/view this content if you trust Microsoft Corporation to make that assertion.
Always trust content from Microsoft Corporation
Yes
No
More Info

Microsoft Security Bulletin MS01-017

Who should read this bulletin: All customers using Microsoft® products.

Technical description: In mid-March 2001, VeriSign, Inc., advised Microsoft that on January 29 and 30, 2001, it issued two VeriSign Class 3 code-signing digital certificates to an individual who fraudulently claimed to be a Microsoft employee. …

Impact of vulnerability: Attacker could digitally sign code using the name "Microsoft Corporation".

Proof-carrying code

PCC certifies code with a condensed formal proof of desired property.

- Checked by client before installation / execution
- Unforgeable, tamper-proof and independent of trust networks
- Proofs may be hard to generate, but are easy to check

Ideally a *certifying compiler* uses types and other high-level source information to create the necessary proof to accompany machine code.

Proof-Carrying Code – George Necula, POPL '97

Safe Kernel Extensions Without Run-Time Checking – Necula+Lee, OSDI '96

Foundational Proof-Carrying Code – Andrew Appel, LICS '01

Inferring resource usage

Resources can include:

- processor time
- heap space
- stack size
- system calls
- disk files
- network bandwidth, *etc.*

There exist strong theoretical results, but applying them is a challenge.

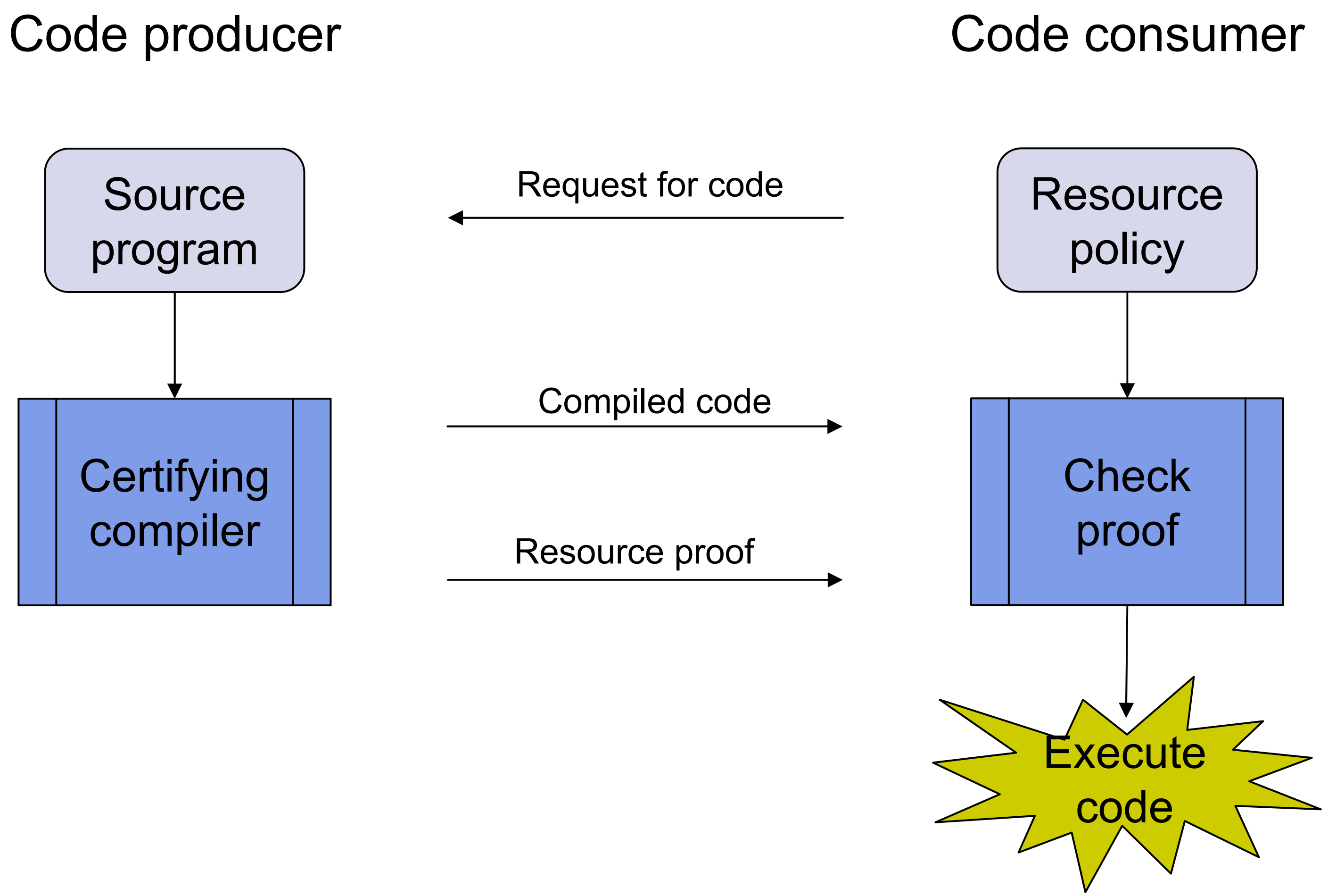
Hofmann – *A type system for bounded space and functional in-place update*

Hofmann+Jost – *Static prediction of heap space usage for first-order functional programs*

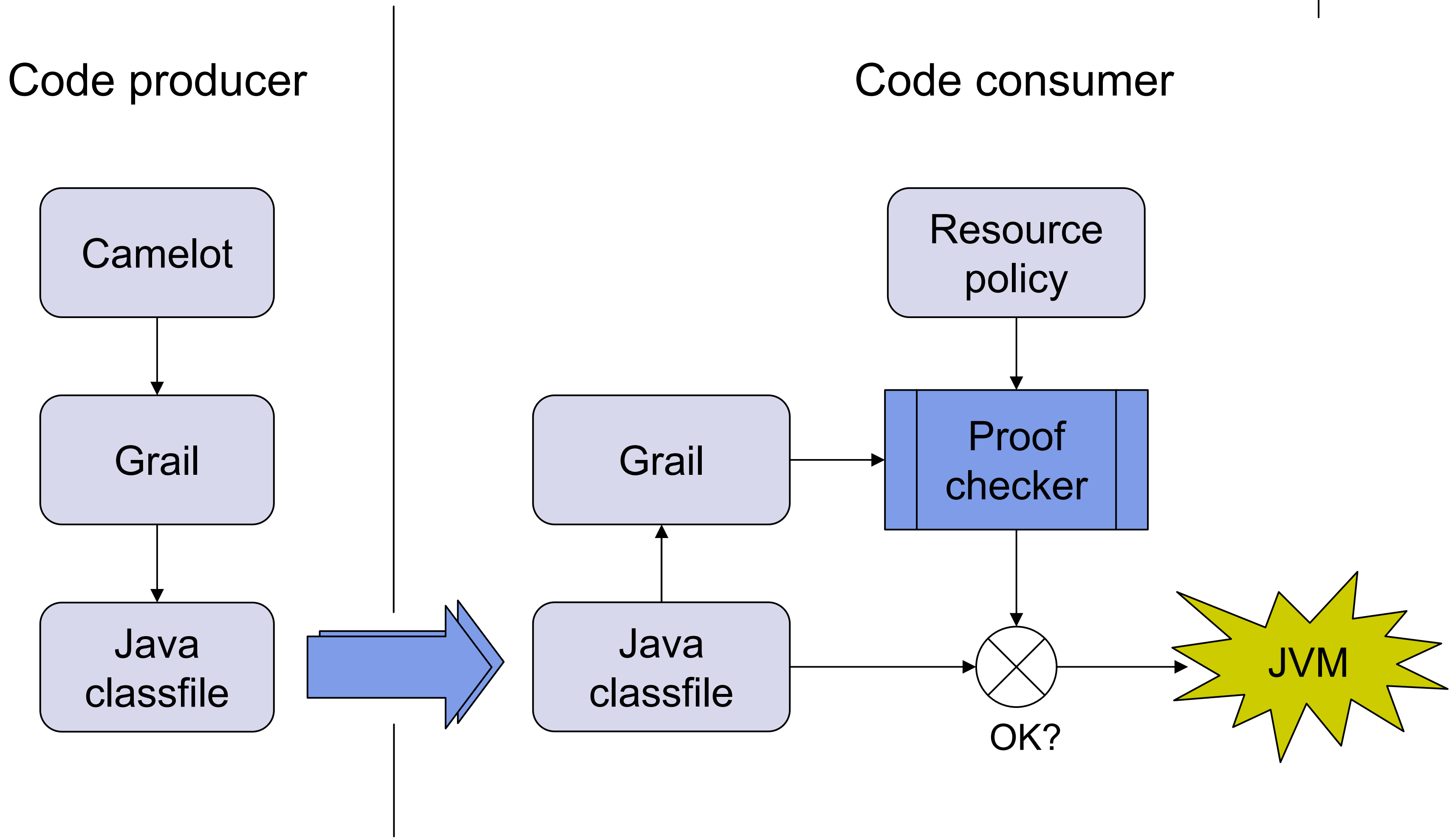
Amadio – *Max-plus quasi-interpretations*

Crary+Weirich – *Resource bound certification*

Architecture



Implementation



GRAIL

Guaranteed Resource Aware Intermediate Language

A key component of the MRG platform is our intermediate language, which needs to be all of the following:

- The target for the *Camelot* compiler
- A basis for attaching resource assertions
- Amenable to formal proof about resource usage
- The format for sending and receiving guaranteed code
- Executable

Grail mediates between all of these roles by having two distinct semantic interpretations, one functional and one imperative.

Functional Grail

Grail has a standard functional semantics:

- Strong static typing
- Call-by-value first-order functions
- Local function declaration
- Mutual recursion
- Lexical scoping of variables and parameters

This simple functional language is the target for the *Camelot* high-level language compiler.

Fibonacci in functional Grail

```
method static int fib (int n) =
  let val a = 0
      val b = 1
      fun loop (int a, int b, int n) =
          let val b = add a b
              val a = sub b a
              val n = sub n 1
           in
              test(n,a,b)
          end
      fun test (int n, int a, int b) =
          if n<=1 then b else loop(a,b,n)
  in
      test(n,a,b)
  end
```

Fibonacci in functional Grail

```
method static int fib (int n) =
  let val a = 0
      val b = 1
      fun loop (int a, int b, int n) =
          let val b = add a b
              val a = sub b a
              val n = sub n 1
           in
              test(n,a,b)
          end
      fun test (int n, int a, int b) =
          if n<=1 then b else loop(a,b,n)
  in
      test(n,a,b)
  end
```

local variable declarations

lexically scoped variables hide outer declarations

local function declarations

mutually recursive function calls

function arguments

Imperative Grail

Grail also has a simple imperative semantics:

- Assignable global variables (registers)
- Labelled basic blocks
- Goto and conditional jumps
- Live-variable annotations

The Grail assembler and disassembler convert this to and from Java bytecodes as an executable binary format.

Fibonacci in imperative Grail

```
method static int fib (int n) =
  let val a = 0
      val b = 1
      fun loop (int a, int b, int n) =
          let val b = add a b
              val a = sub b a
              val n = sub n 1
           in
              test(n,a,b)
          end
      fun test (int n, int a, int b) =
          if n<=1 then b else loop(a,b,n)
  in
      test(n,a,b)
  end
```

Fibonacci in imperative Grail

```
method static int fib (int n) =
  let val a = 0
      val b = 1
      fun loop (int a, int b, int n) =
          let val b = add a b
              val a = sub b a
              val n = sub n 1
           in
              test(n,a,b)
          end
      fun test (int n, int a, int b) =
          if n<=1 then b else loop(a,b,n)
  in
      test(n,a,b)
  end
```

initial assignment to global variables

update global variables

basic blocks

goto and conditional jumps

annotate live variables

Comparing functional and imperative

We can prove a precise correspondence between the two semantics. A Grail method body *mbody* decomposes into (imperative) basic blocks:

$$\mathit{mbody} \;\underset{\text{fun}}{\overset{\text{imp}}{\rightleftarrows}}\; \mathit{blocklist}$$

Theorem: If E is a variable environment and s a matching initial state

$$E =_{var} s \quad \text{where} \quad var = \mathrm{fv}(\mathit{mbody}) = \mathrm{Var}(\mathit{blocklist})$$

then for any final value v:

$$E \vdash_{\text{fun}} \mathit{mbody} \Rightarrow v \quad \text{if and only if} \quad s \vdash_{\text{imp}} \mathit{blocklist} \Rightarrow v$$

where $\vdash_{\text{fun}}$ and $\vdash_{\text{imp}}$ are functional and imperative evaluation respectively.

What makes it work

Definitions of the two semantics $\vdash_{fun}$ and $\vdash_{imp}$ are entirely as expected. The result only holds because we place tight constraints on well-formed Grail.

- No nesting: only one level of local functions
- Functions must include all free variables as parameters
- Tail calls only
- Functions are only applied to values, which must syntactically coincide with the parameter names: `fun f(int x) … f(x)`

Imperative Grail is similarly well-behaved: for example, the stack is empty at all jumps and branches. This is what makes it possible to disassemble JVM classfiles back into Grail again. (metadata helps too)

Free variables and liveness

The functional / imperative match in Grail extends to relating other program analyses. For example, *free* variables for functional terms correspond precisely to the the imperative notion of *liveness.*

let decls in e end —imp→ *bbl*

let decls in e end ←fun— *bbl*

fv(*let decls in e end*) = gen(*bbl*)

dom(*decls*) = kill(*bbl*)

Theorem: A method body satisfies the “no-free-variable” condition on local function declarations *if and only if* the given parameter lists are a valid solution for the liveness dataflow equations.

Linear types and single usage

Beringer [2002] extends classic dataflow analysis to identify variables used exactly once after each update; with applications to memory management and register forwarding in asynchronous processors. For Grail this gives an analysis for the use of variable *x* in basic block *bbl*:

$$\mathrm{uses}_x(bbl) \in \{\, 0 \;\; \top \;\; \bot \;\; 1 \,\}$$

and from this the notion of a variable being *read-once* throughout a method body. The functional counterpart is an intuitionistic linear type system for Grail:

$$\Gamma;\, \Theta, x{:}\sigma \vdash e{:}\tau \quad \Leftrightarrow \quad \mathrm{uses}_x(bbl) = 1$$

Theorem: A method can be typed with variable *x* linear *if and only if* the usage dataflow analysis has a solution where *x* is read-once.

Present status

Progress so far:

- High level language compiler (`camelot`)
- Grail assembler (`gdf`) and disassembler (`gf`)
- Cost model (time, stack, heap, calls; raw and structured)
- Isabelle formulation of Grail operational semantics and cost model
- Sample proofs of time and space bounds
- “Foundational” PCC demonstrator based on Isabelle proof scripts

Current work:

- Hoare logic for Grail implemented in Isabelle (auxiliary variables)
- Isabelle proof that Grail cost model is consistent with JVM

Next tasks and future work

- DIY demonstrator on the web
- Object interworking for Camelot
- Freestanding resource logic for Grail (use separation logic for heap?)
- Proofs generated from high-level resource information (types *etc.*)
- Reduce trusted base (put custom proof checker into Java classloader)

- More examples and applications — suggestions please!

- Other bytecode platforms (.Grail)
- Links to the Grid and e-Science (Java Grande, scientific computation)

EEF Summer School

Global Computing

Edinburgh 7–11 July 2003

- Ian Clarke

  Freenet
- Andrew Gordon

  Security and XML web services
- Martin Hofmann

  Type systems for resource control
- Davide Sangiorgi

  Types and process algebra
- Martin Wirsing

  UML for global computing
- Rocco de Nicola

  KLAIM – a Kernel Language for Agent Interaction and Mobility