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

APPSEM II - Nottingham - Thursday 27 March 2003

Mobile Resource Guarantees

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

The 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)

Context for MRG project

Mobile code and global computation:

- Our target scale is from Java smartcards to desktop applications.
- Self-service code pulled from multiple providers
- Heterogenous clients with irregular resource limitations

How to ensure that programs can still run safely, securely and successfully in this setting? One solution is *proof-carrying code*:

- Certifies program with a compact proof of desired property
- Complements exisiting cryptographic authentication of provider
- Proofs may be hard to generate, but are easy to check

(Necula, Lee, Appel)

Inferring resource usage

Resources can include:

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

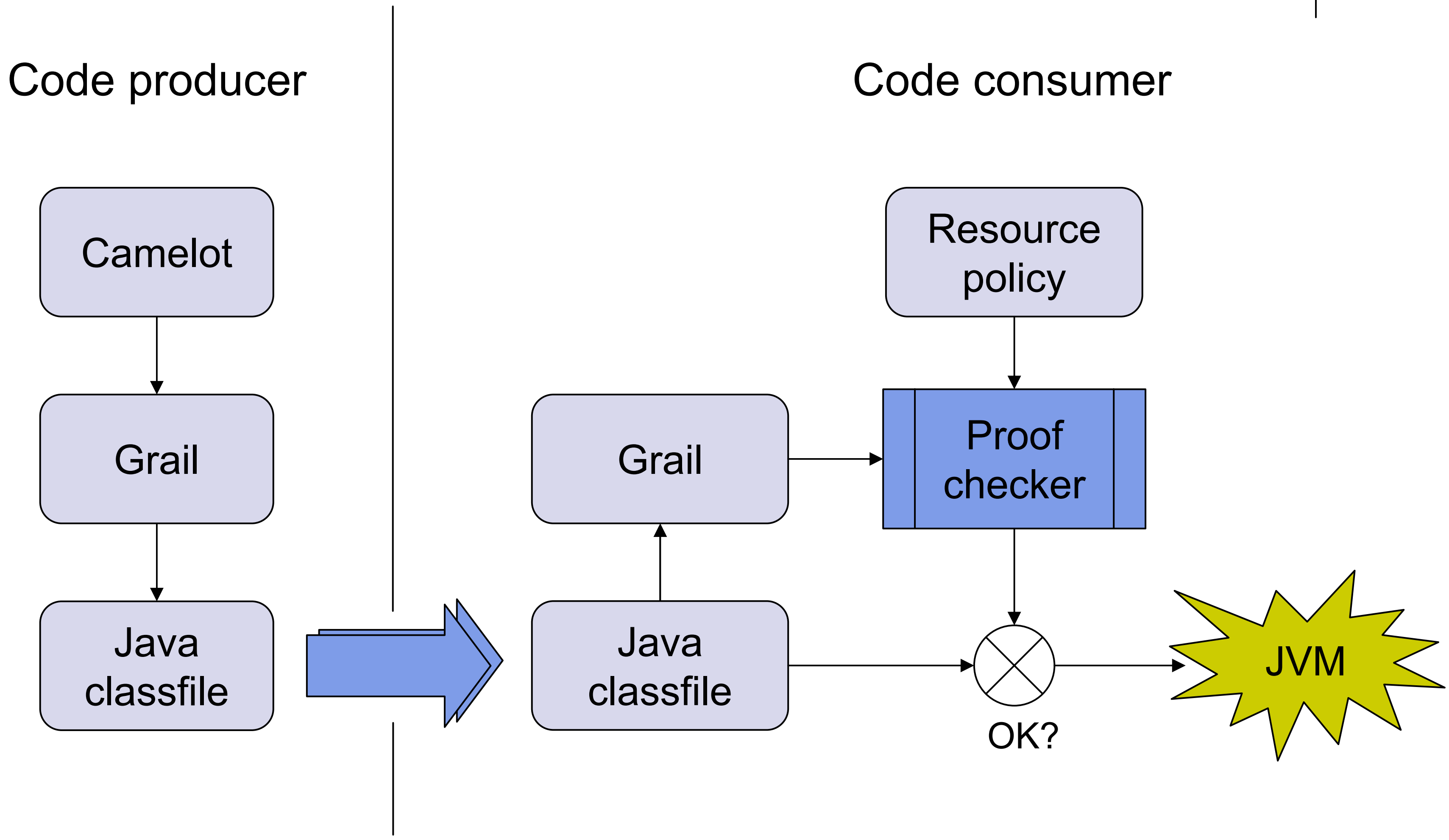
There exist strong theoretical results, but applying them is a challenge.

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*

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.

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

local function declarations

lexically scoped variables hide outer 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

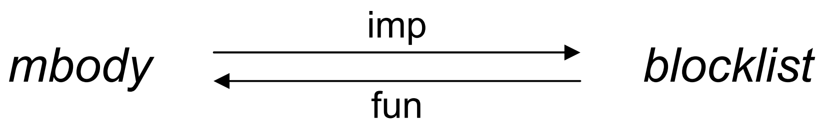
What makes it work

The two semantics really are quite different. Things only work out 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)

Relating functional and imperative



1. If E is a variable environment and s a matching initial state, then for all v, $E \vdash_{\text{fun}} mbody \Rightarrow v$ if and only if $s \vdash_{\text{imp}} blocklist \Rightarrow \text{v}$
2. 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 imperative liveness dataflow equations.
3. A method can be typed with variable *x* linear *if and only if* the imperative usage dataflow analysis has a solution where *x* is read just once after each update (it is “forwardable”).

MRG project progress

Progress so far:

- High level language compiler (`camelot`)
- Grail assembler (`gdf`) and disassembler (`gf`)
- Isabelle formulation of Grail operational semantics and cost model

Working on:

- Resource logic for Grail (use separation logic for heap?)
- Generating proofs from high-level resource information (types *etc.*)

Looking for more examples and applications — suggestions please!