Mobile Resource Guarantees

Don Sannella

Laboratory for Foundations of Computer Science

School of Informatics, University of Edinburgh

Univ. of Edinburgh: David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Kenneth MacKenzie, Alberto Momigliano, Matthew Prowse

LMU Munich: Martin Hofmann, Hans-Wolfgang Loidl, Olha Shkaravska

Mobile Resource Guarantees

MRG is a joint Edinburgh / LMU Munich project funded for 2002–2005 by the European Commission's *Global Computing* pro-active initiative.

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 so far, and in particular our *GRAIL* intermediate language, resource types, and bytecode logic.

Roadmap

1. Global computing
2. Proof-carrying code
3. … for resource certification
4. Overview of progress on MRG

Global computing

We now have networked access to vast computational resources: hardware, software, data

The network(s) is/are planet-wide and dynamically changing, and location of resources (at least, Europe vs Australia) matters

The availability and responsiveness of these resources is unpredictable and uncontrollable; no accurate global information is available

Global computing = an emerging computational paradigm in which these resources are flexibly exploited by mobile agents

“Programming the internet”, but more than that

Global computing

Dominant concerns of traditional computing: representing and manipulating data efficiently

Dominant concerns of global computing: security, reliability, robustness, failure modes, locality, control of resources, coordination, interaction

Related to: distributed computing, peer-to-peer systems, ubiquitous computing, the Grid, agents, active networks, etc.

Global computing

Dominant concerns of traditional computing: representing and manipulating data efficiently

Dominant concerns of global computing: security, reliability, robustness, failure modes, locality, control of resources, coordination, interaction

Related to: distributed computing, peer-to-peer systems, ubiquitous computing, the Grid, agents, active networks, etc.

Roadmap

1. Global computing
2. Proof-carrying code
3. … for resource certification
4. Overview of progress on MRG

Authentication for mobile code

Java

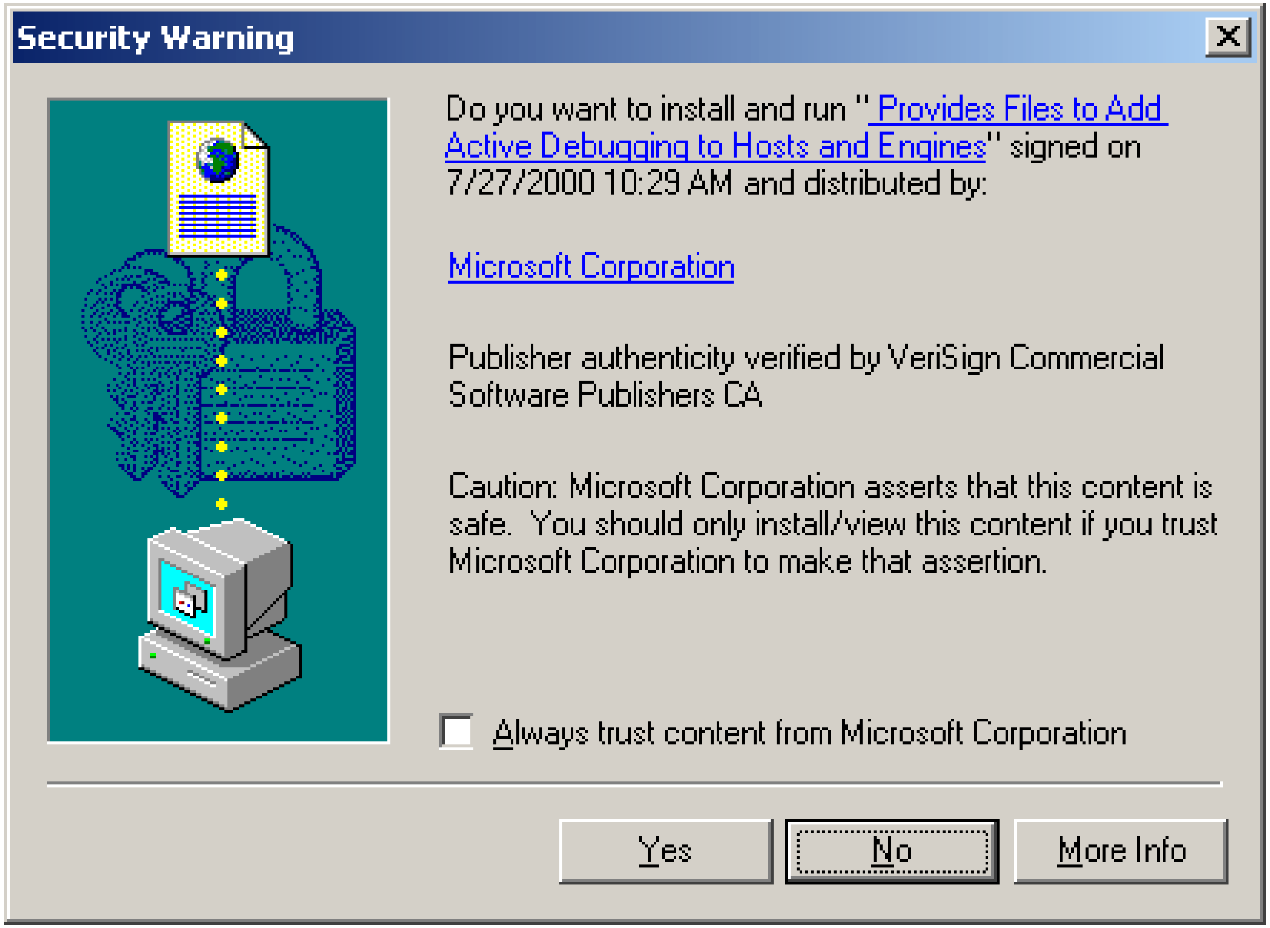
- Originally, Java used a *sandbox* model, where all remote code was wholly untrusted.
- In version 1.2 this moved to more finely grained *security policies* which can be specified using cryptographic signatures on code.

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.

However, crypto signatures say nothing about the code itself, only its supplier.

In Microsoft I trust



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 a desired property.

- Checked by client before installation / execution
- Proofs may be hard to generate, but are easy to check
- Independent of trust networks: unforgeable, tamper-evident

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

Roadmap

1. Global computing
2. Proof-carrying code
3. … for resource certification
4. Overview of progress on MRG

Resource-bounded computation

A user of a handheld device, wearable computer, or smart card wants to know that a downloaded application will definitely run within the limited amount of memory available.

A provider of distributed computing power may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption.

Third-party software updates for mobile phones, household appliances, or car electronics should come with a guarantee not to set system parameters beyond manufacturer-specified safe limits.

Inferring resource usage

Resources can include:

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

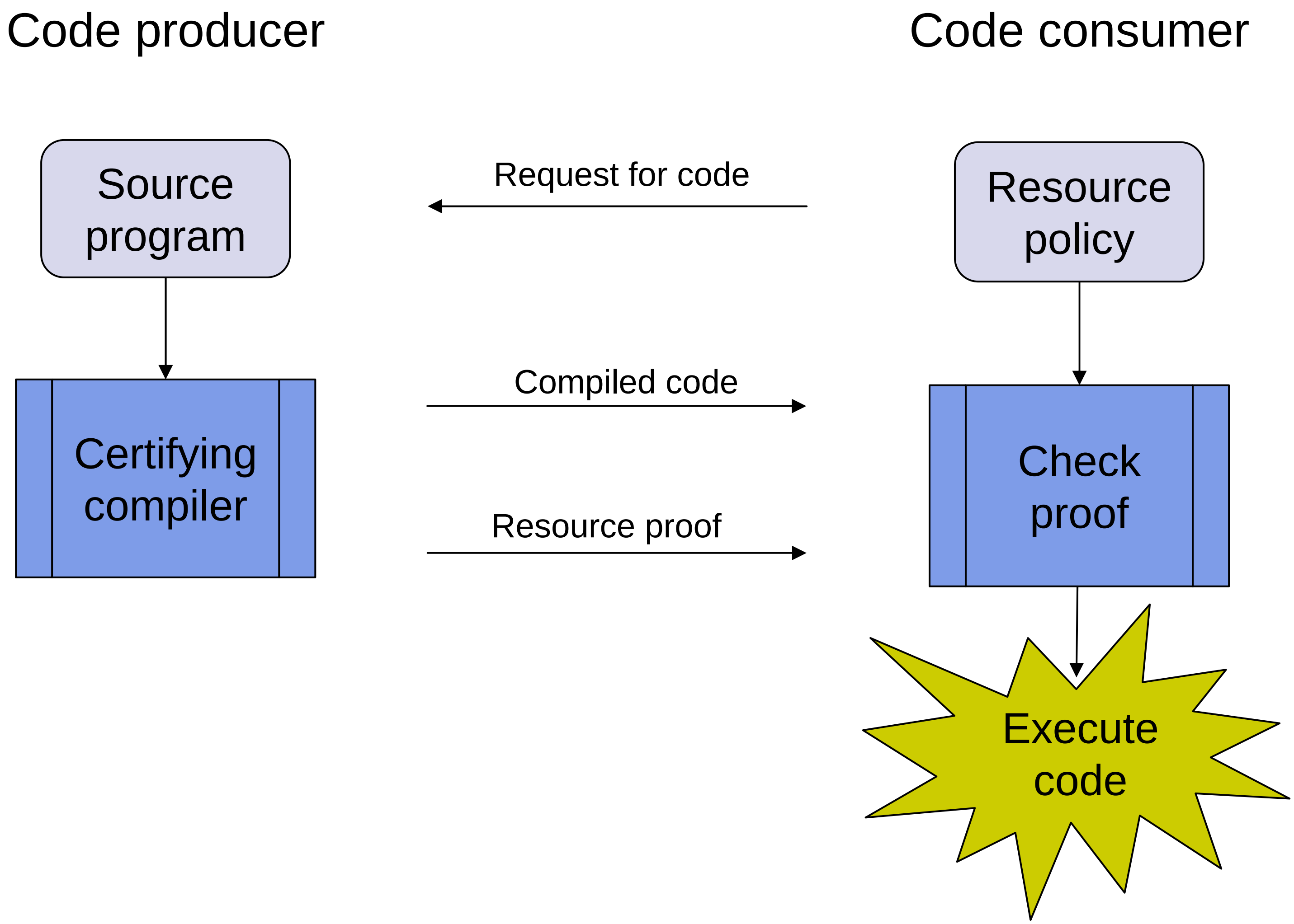
There exist strong theoretical results, but applying them is a challenge.

We have been concentrating mainly on heap space, so far.

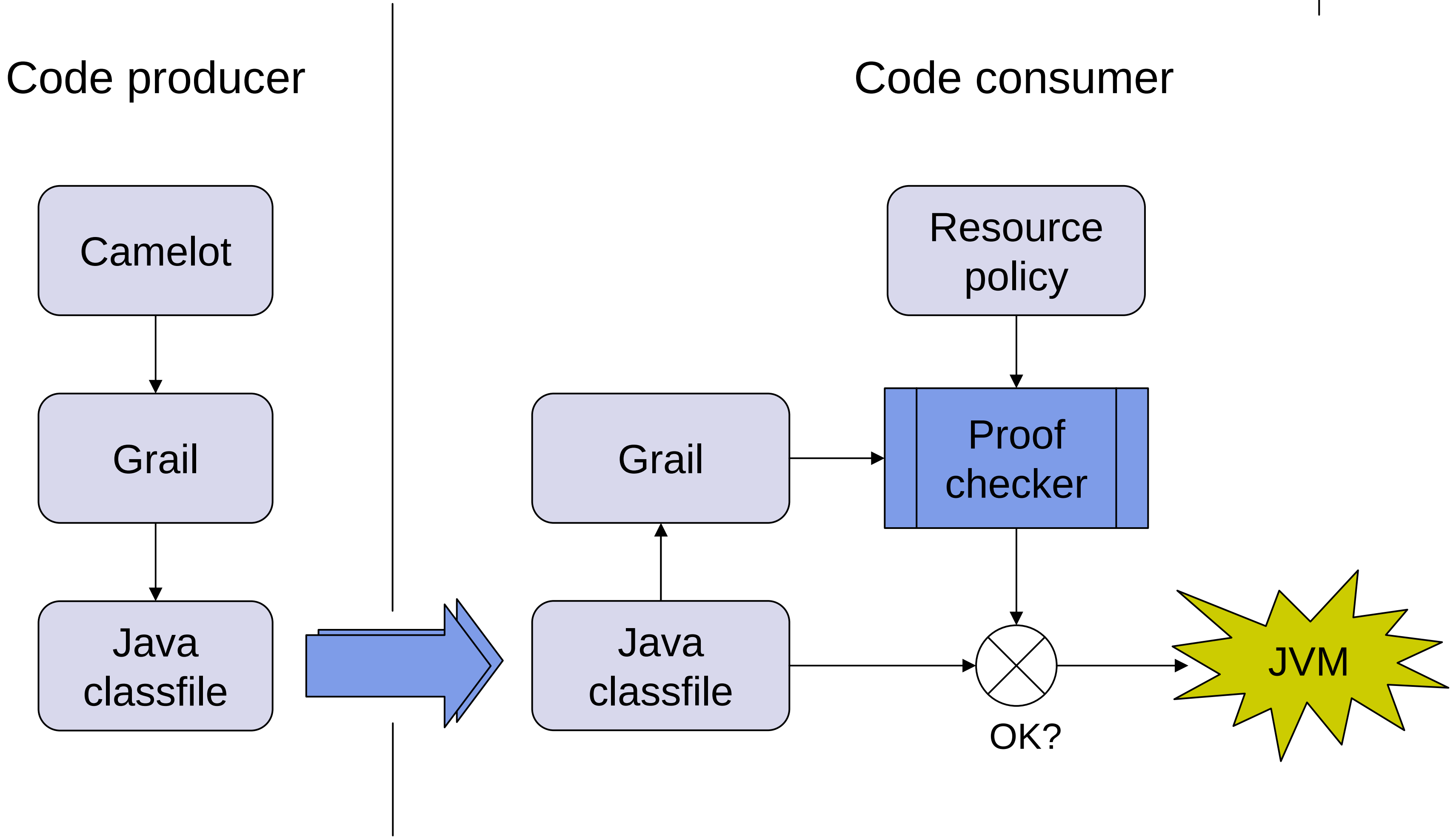
Hofmann – *A type system for bounded space and functional in-place update*

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

Architecture



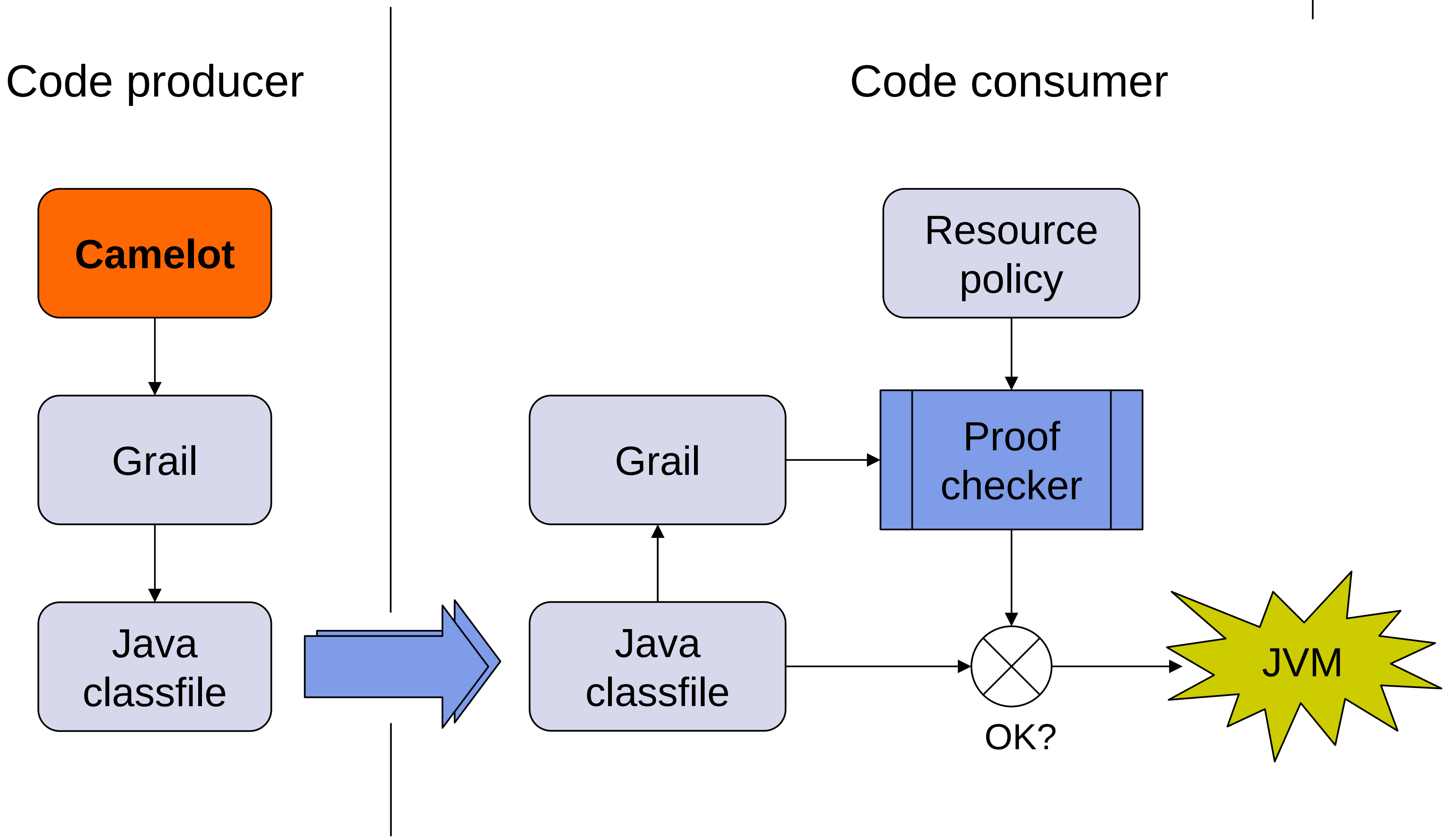
Implementation



Roadmap

1. Global computing
2. Proof-carrying code
3. … for resource certification
4. Overview of progress on MRG

Implementation



Camelot

A high-level call-by-value functional language based on OCaml

- Polymorphism, constructor-based datatypes, pattern-matching
- First-order functions only, to avoid heap-allocated closures
- Objects for access to the Java class hierarchy
- Constructs for explicit control of heap usage
- A resource typing system to enforce linear (i.e. affine) usage of heap-allocated objects
- Inference of heap space usage bounds
- Further extensions ongoing: restricted higher-order, threads.

Resource types in Camelot

```
Cons(-,-)   : ‘a * ‘a list      -> ‘a list

rev : ‘a list * ‘a list -> ‘a list
```

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

Resource types in Camelot

```
Cons(-,-)@- : ‘a * ‘a list * <> -> ‘a list

rev : ‘a list * ‘a list -> ‘a list


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

Resource types in Camelot 2

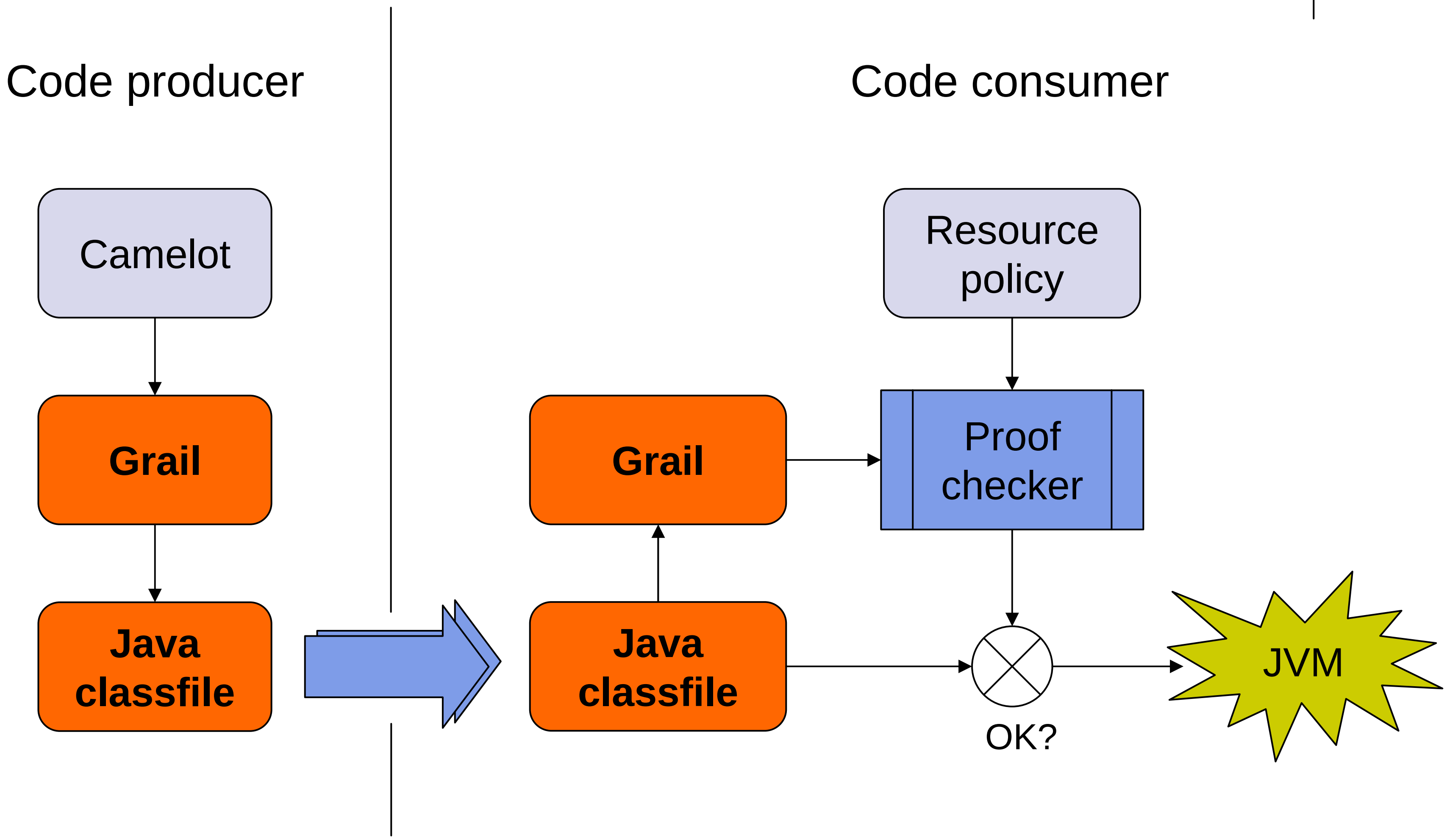
```
insert : int * int list * <> -> int list

let insert n l d =
  match l
    with Nil -> Cons(n,Nil)@d
       | Cons(h,t)@d’ -> if n <= h then(Cons(n,Cons(h,t)@d’)@d
                         else Cons(h,insert n t d)@d’

sort : int list -> int list

let sort l =
  match l
    with Nil -> Nil
       | Cons(h,t)@d -> insert h (sort t) d
```

Implementation



GRAIL
Guaranteed Resource Aware Intermediate Language

Our intermediate language 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 certified 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

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

Comparing functional and imperative

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

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

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

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

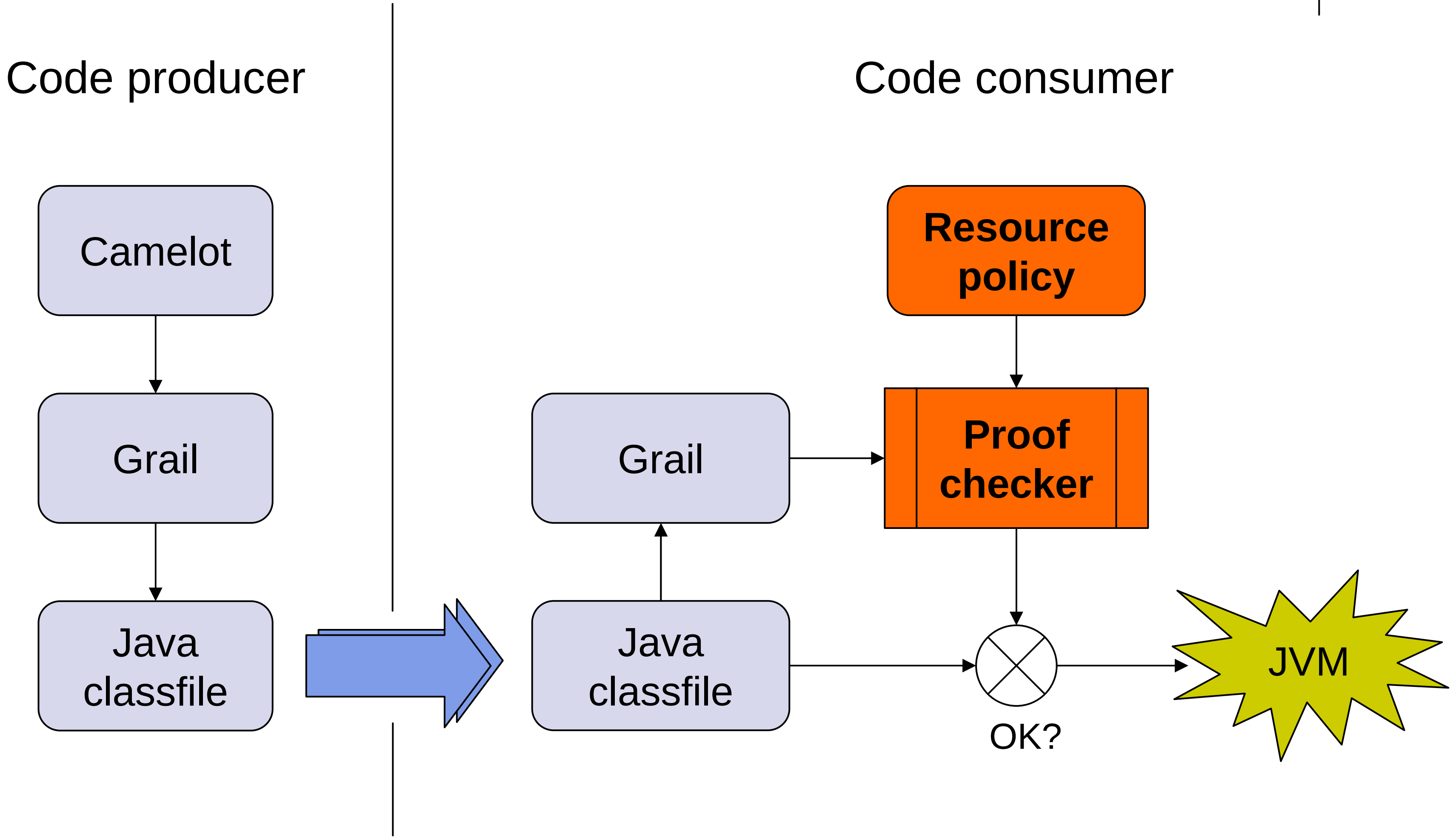
What makes it work

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 functional 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.

Implementation



Bytecode logic of resources

- A VDM-style logic, without pre-conditions
- Assertions are for *partial* correctness:
  - Separate consideration of termination argument
- Assertions are given for functional Grail (almost bytecode)
- Formalised in Isabelle/HOL using a shallow embedding
- Sound and (relative) complete
- Certificates are, for now, Isabelle proof scripts
  - A somewhat large trusted code base!
  - For small devices, use *off-device pre-verification* (Java CLDC)

Operational semantics & assertions

- We give a big-step operational semantics:

  `E ⊢ h,e ⇓ (h′,v,r)`

  E is an environment, h and h' are heaps, v is a value and

  r = (ticks, callcount, invokecount, invokedepth).

- The logic is closely related: an assertion P specifies possible executions for an expression:
  - e : P(h,h',v,r)

    if and only if

    ∀ E,h,h',v,r. `E ⊢ h,e ⇓ (h′,v,r)` implies P(h,h',v,r)

  We prove both directions in the formalisation.

Example

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

▸ `call rev` : SpecRev

where SpecRev specifies consumption of:

- 0 heap space
- L+1 function calls
- 31L+11 clock ticks

where L = length `l`

Present status

- High level language compiler (`camelot`)
- Grail assembler (`gdf`) and disassembler (`gf`)
- Cost model (time, stack, heap, calls)
- VDM-style logic for Grail, implemented in Isabelle/HOL
- PCC demonstrator based on Isabelle proof scripts
- Various resource type systems for heap space
- Resource type inference for heap space

Current work:

- Proof certificates generated from resource types
- Resource type inference for stack space

Thank you!

Mobile
Resource
Guarantees

http://www.lfcs.ed.ac.uk/mrg

Certifying compiler

Resource types can be inferred automatically for first-order functions

Proof certificates can be generated (automatically?) from source code and resource type

- relies on higher-level proof rules
- … which are derived rules in the bytecode logic

Future Project: MRG and the Grid

- Camelot and Grail programs can run in very resource-constrained environments such as the KVM. What is the relevance to the Grid?

Future Project: MRG and the Grid

- Camelot and Grail programs can run in very resource-constrained environments such as the KVM. What is the relevance to the Grid?
- Grid service providers need to schedule competing requests for access to resources. There is a specification language (RSL) for resources, but …

```
&(reservation-type=compute) (start-time="10:30pm")
(duration="1 hour") (nodes=32)
```

- Mobile code seems perfect for the Grid: with 25Kb of code and 1Pb of sky survey data it is infeasible to ship the data to the code.
- We will try to transfer MRG results to Java, using ESC/Java, to produce much more precise resource bound specifications.