

# Automatic Certification of Resource Consumption

Alberto Momigliano

Laboratory for Foundations of Computer Science

University of Edinburgh

In collaboration with: see credits at the end of the talk

Work carried out in the EU-project "Mobile Resource Guarantees" (MRG),

IST-2001-33149

Comète-Parsifal Seminar, March 7th, 2005

## MRG: PCC infrastructure for resource-related properties

---

- MRG is a joint University of Edinburgh / LMU Munich project funded for 2002-2005 by the European Commission's pro-active initiative in Global Computing.
- The aim is to endow mobile code with independently verifiable certificates describing resource requirements, following the *proof-carrying code* paradigm.
- Applications with resource considerations: portable devices (phones, PDA's,...), Smartcards, embedded processors (car electronics,...), satellites, GRID services,...
- Example resources: memory (heap & stack), time, energy, network bandwidth, parameter values of system calls
- PCC: code consumer requires transmitted program to come with verifiable proof that his resource policy is fulfilled
- Approach (certifying compilation): translation from user language into machine language derives independently verifiable certificates

## Outline

---

- Architecture of MRG
- Syntax and semantics of Grail
- Grail's Program Logic
- Derived Assertions
- Web demo
- Conclusions

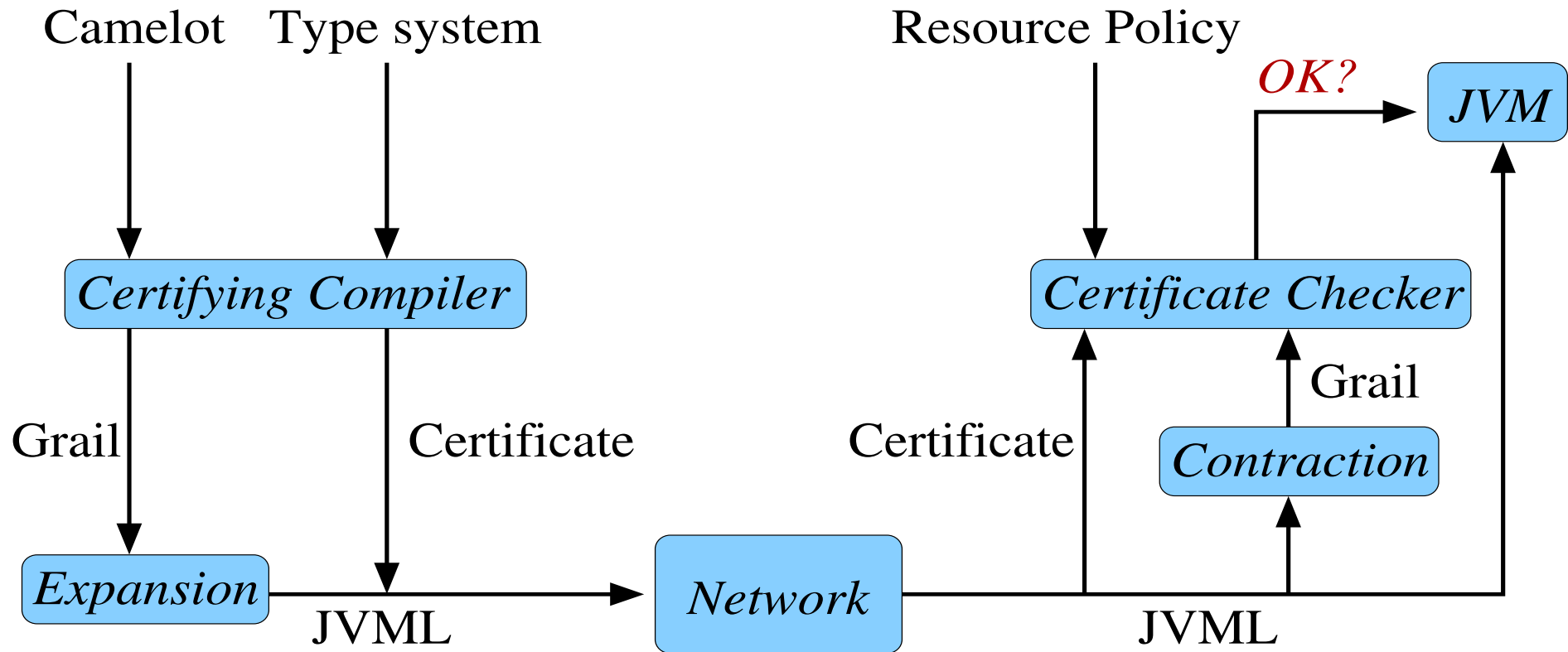
## Components of MRG

---

- We 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 JVMIL.
- We use an abstract **cost model** for the JVM which counts instructions and measures stack and heap sizes.
- Costs are calculated using a **annotated operational semantics** for Grail, reflecting the expansion into JVMIL.
- **Grail Logic** is a program logic which can express resource assertions about the operational semantics.
- Camelot has a **resource type inference system**, which is used to produce proofs in a **logic of derived assertions**.
- The annotated semantics, logics, and meta-theorems have all been formalised in **Isabelle**, and Isabelle proof scripts are used as our proof transmission format.

# MRG architecture

---



Works because of reversible expansion of Grail into JVML subset

# Camelot

---

Camelot: ML-like first-order functional language (polymorphism, no references)

- 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)
```

- Notation @\_ indicates destructive pattern match
- Whole program compilation where each Camelot function yields one JVM method
- Compilation includes an explicit memory manager (freelist)

Wish to certify memory consumption of compiled output.

## Program analysis, certification & proof checking

---

```
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)
```

- Memory consumption inferred from program annotations using a type system
- Result: `ins` consumes one memory cell, independent from actual input, `sort` does not consume any memory (in-place)
- In general: memory consumption expressed relative to size of input
- PCC-certificate: encoding of the result of the type inference in a program logic
- Certificate bundled with program for transmission
- JVM at consumer side uses modified class loader (security manager) that checks certificate in Isabelle before executing program

## PCC: us and them

---

### Existing approaches:

- Classic PCC: trusted special-purpose proof systems for proving light-weight properties of machine code (memory safety)
- Foundational PCC: operational model (processor) formalised in higher-order logic that is built on top of theorem prover Twelf, use Twelf proof terms as certificates

### MRG:

- Formalise *instrumented* operational semantics of (virtual) machine language
- Use a general-purpose program logic (sound, complete & expressive, little automation)
- Derive special logics (interpreted type systems) in theorem prover

## Grail: Characteristics

---

- Combine OO-aspects of bytecode (fields, methods) with (impure) low-level functional language
- Extends Appel-Kelsey-correspondence to machine level
- Functional view: first-order functions; no nesting; all free variables in parameters; applications only to values.
- Imperative view: easily convertible into various virtual machines formats
- registers = variables, jumps = tail-calls
- Coincidence between functional and imperative views makes conversion reversible
- Emitted bytecode is highly structured (Leroy's conditions)

## Syntax of Grail

---

- A Grail program is a list of *methods* each containing a list of tail-recursive *functions*.

$$\begin{aligned} e \in \text{expr} & ::= \text{null} \mid i \mid x \mid \text{prim } p \ x \ x \\ & \mid \text{new } c \ [\overline{t_i := x_i}] \\ & \mid x.t \mid x.t := x \\ & \mid \text{let } x = e \text{ in } e \mid e; e \\ & \mid \text{if } x \text{ then } e \text{ else } e \\ & \mid \text{call } f \mid c.m(\overline{a}) \end{aligned}$$
$$a \in \text{args} ::= x \mid \text{null} \mid i$$

- Whole development formalized in Isabelle/HOL:
- named syntax,
- program encoded using global tables (functions and methods),
- op. semantics based on (finite) maps:

## Grail: resource-instrumented operational semantics

---

- Based on (impure) big-step functional view:

$$E \vdash h, e \Downarrow (h', v, p)$$

where  $r$  is a *resource value* in some *resource algebra*  $\mathcal{R}$ .

- Moreover, the resources  $r$  are a purely “non-invasive” annotation on an ordinary operational semantics; evaluation of an expression is not affected by the resources consumed in subexpressions.
- The resource algebra has families of operations for each of the syntactic constructs of Grail...
- A resource algebra  $\mathcal{R}$  has a carrier set  $R$  consisting of *resource values*  $r \in R$ , with:
  - For the atomic expressions, families of constants  $\mathcal{R}^{\text{null}} \in R$ , etc.
  - For compound expressions, families of operations, e.g.  $\mathcal{R}_x^{\text{let}} \in R \times R \rightarrow R$ .
- JVM case:  $R$  consists of quadruples:

$$r = (\text{clock}, \text{callc}, \text{invkc}, \text{invkdepth})$$

- Stack usage is approximated; heap usage calculated as the difference  $\text{size}(h') - \text{size}(h)$ .

## Operational semantics

---

$$\frac{E\langle x \rangle = \text{Ref } l}{E \vdash h, x.t \Downarrow (h, h(l).t, \mathcal{R}^{\text{getf}}(x, t))} \quad (\text{GETF})$$

$$\mathcal{R}^{\text{getf}}(x, t) = \langle 2000 \rangle.$$

$$\frac{E \vdash h, e_1 \Downarrow (h_1, w, p) \quad w \neq \perp \quad E\langle x := w \rangle \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, \text{let } x = e_1 \text{ in } e_2 \Downarrow (h_2, v, \mathcal{R}^{\text{let}}(x, p, q))} \quad (\text{LET})$$

$$\mathcal{R}^{\text{let}}(r_1, r_2) = (1 + \max(r_1, r_2))$$

$$\frac{E \vdash h, f_{\text{body}} \Downarrow h', v, r}{E \vdash h, \text{call } f \Downarrow h', v, \mathcal{R}_f^{\text{call}}(r)}$$

$$\mathcal{R}_f^{\text{call}}(t, c, i, d) = (t + 1, c + 1, i, d)$$

## Other resource algebras (current work)

---

- Resource algebras usefully generalise the specific case to allow richer resource/security policies to be expressed. Examples include:
  - *parameter limit flags* set by parameter limit policies; here simply  $\mathcal{R} = \{\text{true}, \text{false}\}$ .
  - *traces of method invocation sequences*, so e.g.  $\mathcal{R} = \{m^*\}$  where  $m$  ranges over method names.
  - *read-write effects on heap locations*, where  $\mathcal{R} = \{\langle \text{Rd}, \text{Or}, \text{RdWr} \rangle\}$  for  $\text{Rd}, \text{Wr}, \text{RdWr} \subseteq \text{Locations}$ . Other e.g.s: live variables, complete traces of heaps during execution, ...
- For some examples, additional indices/sorts are needed for the environment (stack) and heap, to extract or examine values.
- Further algebraic structure on  $\mathcal{R}$  is perhaps useful and is currently under investigation. Current idea: a monoid with semi-lattice structure: composition of monoid  $+$  is composition of resources;

## Program logic I

---

- Recent reappraisal of program (Hoare) logics: embeddings in theorem prover (Kleymann, Nipkow), Separation logics (Reynolds, O'Hearn), Java verification (Jacobs, de Boer, von Oheimb)
- Embedding a la Kleymann: deep embedding of language, shallow embedding of assertions, with soundness and (relative) completeness formally proven in theorem prover
- Pragmatic issue: meta-theoretic investigation vs program verification (automation). In MRG-PCC both issues are important!
- Judgements take the form  $G \triangleright e : P$ 
  - $e$  is a Grail expression;
  - $G$  is a set of assumptions context used for storing assumptions for recursive methods and functions;
  - $P$  is an assertion, i.e. a predicate in the meta-logic
  - Assertions are simply predicates over semantic values:

$$P[E, h, h', v, r]$$

relating the environment, initial and final heaps, the result and the resource value.

## Program logic II: proof rules

---

- No auxiliary variables (usage of pre-heap inspired by hooked variables in VDM)
- Judgements interpreted as partial “correctness” statements: validity  $\models e : P$  defined as

$$\forall E \ h \ h' \ v \ p. \ (E \vdash h, e \Downarrow (h', v, p) \longrightarrow P[E, h, h', v, r])$$

- Termination considered orthogonal

$$\frac{}{G \triangleright x.t : \lambda E \ h \ h' \ v \ p. \exists l. \ E \langle x \rangle = \text{Ref } l \wedge h' = h \wedge v = h'(l).t \wedge p = \mathcal{R}^{\text{getf}}(x, t)} \quad (\text{VGETF})$$

$$G \triangleright e_1 : P_1 \quad G \triangleright e_2 : P_2$$

$$\frac{}{G \triangleright \text{let } x = e_1 \text{ in } e_2 : \lambda E \ h \ h' \ v \ p. \exists p_1 \ p_2 \ h_1 \ w. \ P_1[E, h, h_1, w, p_1] \wedge w \neq \perp \wedge P_2[E \langle x := w \rangle, h_1, h', v, p_2] \wedge p = \mathcal{R}^{\text{let}}(x, p, q)} \quad (\text{VLET})$$

- Much simpler than Hoare-style logic (variable update in precondition)
- Structural and admissible rules: context lookup, rule of consequence, CUT.

## Program logic III: soundness & completeness

---

- Soundness proven as usual, by relativised validity and induction on height of derivations
- “Relative” completeness (Cook, Aczel): in rule of consequence, the implication only needs to *hold* rather than being *derivable*: incompleteness of HOL is inherited by program logic since language of assertions is not formalised
- Completeness proven by induction over program structure, by defining strongest specifications (most general triples)
- Both theorems have been proven in our mechanised formalization. For details, see our paper in TPHOLs '04.
- A Grail program consists of a number of mutually methods/functions. To prove that each method and function satisfies a specification/invariant, we use a *specification table* SPEC which associates an assertion to each method/function.
- A context  $G$  is *table consistent* (“good”) for a program if it contains assumptions only of the form used in the procedure rules, and moreover the body of each procedure satisfies the claimed specification.

## Program logic IV: example specification (insertion sort)

---

$$\begin{aligned} \text{insSpec} &\equiv \text{SPEC List ins } [a_1, a_2] = \\ &\quad \lambda E h h' v p . \forall i r n X . \\ &\quad (E \langle a_1 \rangle = i \wedge E \langle a_2 \rangle = \text{Ref } r \wedge h, r \models_X n \\ &\quad \longrightarrow |dom(h)| + 1 = |dom(h')| \wedge p \leq \langle (A n + B) (C n + D) (E n + F) (G \end{aligned}$$

$$\begin{aligned} \text{sortSpec} &\equiv \text{SPEC List sort } [a] = \\ &\quad \lambda E h h' v p . \forall i r n X . \\ &\quad (E \langle a \rangle = \text{Ref } r \wedge h, r \models_X n \longrightarrow |dom(h)| = |dom(h')| \wedge p \leq \dots) \end{aligned}$$

Lemma:  $\text{insSpec} \wedge \text{sortSpec} \longrightarrow \triangleright \text{List.sort}([xs]) : \text{SPEC List sort } [xs]$

- $h, r \models_X n$  defined inductively, introduces case-splits during verification
- Proof rules contain existentials over intermediate heaps and instrumentations
- $\rightsquigarrow$  automatic proof search impractical (and not desirable in MRG) even after applying all proof rules (VCG): automation by compiler difficult
- Certificate Generation: exploit program structure and compiler analysis by proving properties that are more closely related to the type system

## Insertion sort: compiler output

---

```
method static public List ins(int a, D l) = ...D.make(a, null)...
```

```
method static public List sort(D l) =
```

```
  if l = null then null
```

```
    else let h = l.HD in let t = l.TL in let _ = D.free(l) in
```

```
      let l = List.sort(t) in List.ins(h, l)
```

... plus code for memory management and runtime environment methods

- **D.make**(...): takes object from freelist, or calls **new**
- **D.free**(x): inserts object into freelist
- **D.main**(l): constructs initial freelist, calls `List.sort(s2i(l))`

We wish to verify that

- any memory allocation throughout an invocation of **main** is performed during the initial construction of the freelist, and in particular that
- during the execution of `List.sort(l)`, all invocations of **make** are executed on a non-empty freelist, i.e. no call to **new** is performed

## Type-based analysis of Camelot programs

---

Type system by Hofmann and Jost (POPL 2003):

- Input: program containing a function **start**: `string list -> unit`  
Output: a *linear function*  $s$  such that **start**( $\perp$ ) will not call **new** when evaluated in a heap  $h$  where
  - $\perp$  points in  $h$  to a linear list of some length  $n$
  - the freelist which forms a part of  $h$  is well-formed
  - the freelist does not overlap with  $\perp$
  - the freelist has length not less than  $s(n)$
- How does this work?
  - Annotate types with freelist annotations for each constructor:  $\mathbf{L}(k)$
  - Judgements  $\Gamma, n \vdash e : T, m$  include information about *initial* and *final* size of freelist
  - Express final size of freelist as function of the size of the output
  - Complement this type system with some method for preventing deallocation of live cells (linear typing, usage aspects, layered sharing, . . .)

## What is certificate generation?

---

- Verify the soundness of the type system w.r.t. the Camelot compilation by
  - interpreting the judgements in the program logic, using basic predicates about freelist representation and length, disjointness conditions of data-structures, *footprint* of program fragments
  - formally proving (in Isabelle/HOL) derived proof rules in the base logic
- Formulate the rules such that automated verification is possible
  - simple side conditions, no  $\exists$ -instantiations, syntax-directed;
  - provided that results of the compile-time analysis are communicated as method-level specifications (invariants)

$$\text{List.ins} \quad : \quad 1, \mathbf{l} \times \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0$$
$$\text{List.sort} \quad : \quad 0, \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0$$

- Fixed assertion format  $\llbracket \mathbf{U}, n, [\Delta] \blacktriangleright \mathbf{T}, m \rrbracket$

$$\text{List.ins} \quad : \quad \llbracket \{\mathbf{a}, \mathbf{l}\}, 1, [\mathbf{a} : \mathbf{l}, \mathbf{l} : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0 \rrbracket$$
$$\text{List.sort} \quad : \quad \llbracket \{\mathbf{l}\}, 0, [\mathbf{l} : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0 \rrbracket$$

## Proof rules

---

- LFD rule ( $\overline{\text{Let}}$ ):

$$\frac{\Gamma_1, n \vdash e_1 : A, k \quad \Gamma_2, x : A, k \vdash e_2 : B, m}{\Gamma_1 \Gamma_2, n \vdash \text{let } x = e_1 \text{ in } e_2 : B, m}$$

- Note linearity condition for eliminating deallocation of live cells
- Certificate logic: linear context implemented in two components
- Proof rule ( $\overline{\text{Let}}$ ):

$$\frac{G \triangleright e_1 : [\mathcal{U}_1, n, [\Gamma] \blacktriangleright S, k] \quad G \triangleright e_2 : [\mathcal{U}_2, k, [\Gamma, x : S] \blacktriangleright T, m]}{G \triangleright \text{let } x = e_1 \text{ in } e_2 : [\mathcal{U}_1 \cup (\mathcal{U}_2 \setminus \{x\}), n, [\Gamma] \blacktriangleright T, m]} \quad \mathcal{U}_1 \cap (\mathcal{U}_2 \setminus \{x\}) = \emptyset$$

- Proof rules are expressed at a level where program variables occur (affinely) linear
- Atomic rules for (destructive and non-destructive) match-statements and for invocations of **make**
- Only the verification of the wrapper (uniform for all programs) needs to unfold the interpretation into the core logic

## Certificates and automated verification

---

Producer-generated certificate:

- Content: method-level specifications in derived-assertions form
- Representation: Isabelle/HOL script that invokes a standard tactic **proveMe**

Consumer side:

- Tactic **proveMe** that
  - invokes derived proof rules (syntax-directed) and
  - discharges side conditions (set inclusions, arithmetic (in-)equalities).
  - Methods verified once, combination for mutual recursion via cut rule and parameter adaptation
  - Functions (basic blocks) verified once, via optimised treatment of merge points that combines imperative (dominator property) and functional (function parameters) viewpoints
  - Currently verified programs: functions over lists and trees (append, flatten, insertion sort & heap sort, ...)
  - On-going generalization to algebraic data-type.

## Discussion

---

### Future work:

- Generalise existing system of derived assertions (sharing, usage-aspects, separation), and evaluate on bigger examples
- Extract stand-alone proof checker
- Derive specialised logics and certificate generation for other resources: frame stack, time, limits and separation conditions on method parameters

### Conclusion:

- MRG-motto: certificate generation by interpreting type-systems in program logic
- Presented expressive program logic for low-level language
- Chain of abstractions: operational semantics  $\rightarrow$  general program logic  $\rightarrow$  derived specialised logics with automation
- Development backed up by implementation in Isabelle/HOL
- Sweet spot in debate “Classic vs. Foundational” PCC:
  - $\rightsquigarrow$  Negotiation between proof size and TCB size

## Credits

---

Numerous researchers and students have contributed to work on the MRG project, including:

- Don Sannella, Ian Stark, Stephen Gilmore, Martin Hofmann, David Aspinall;
- Kenneth MacKenzie, Lennart Beringer, Michal Konečný;
- Hans-Wolfgang Loidl, Olha Shkaravska;
- Matthew Prose, Nicholas Wolverson, Laura Korte;
- Robert Atkey, Steffen Jost;
- Robert Amadio.