Automatic Certification of Resource Consumption

Alberto Momigliano
Laboratory for Foundations of Computer Science
University of Edinburgh

Joint work with Lennart Beringer, Martin Hofmann and Olha Shkaravska

EU-project ”Mobile Resource Guarantees” (MRG), IST-2001-33149

LPAR’04, March 17th, 2005

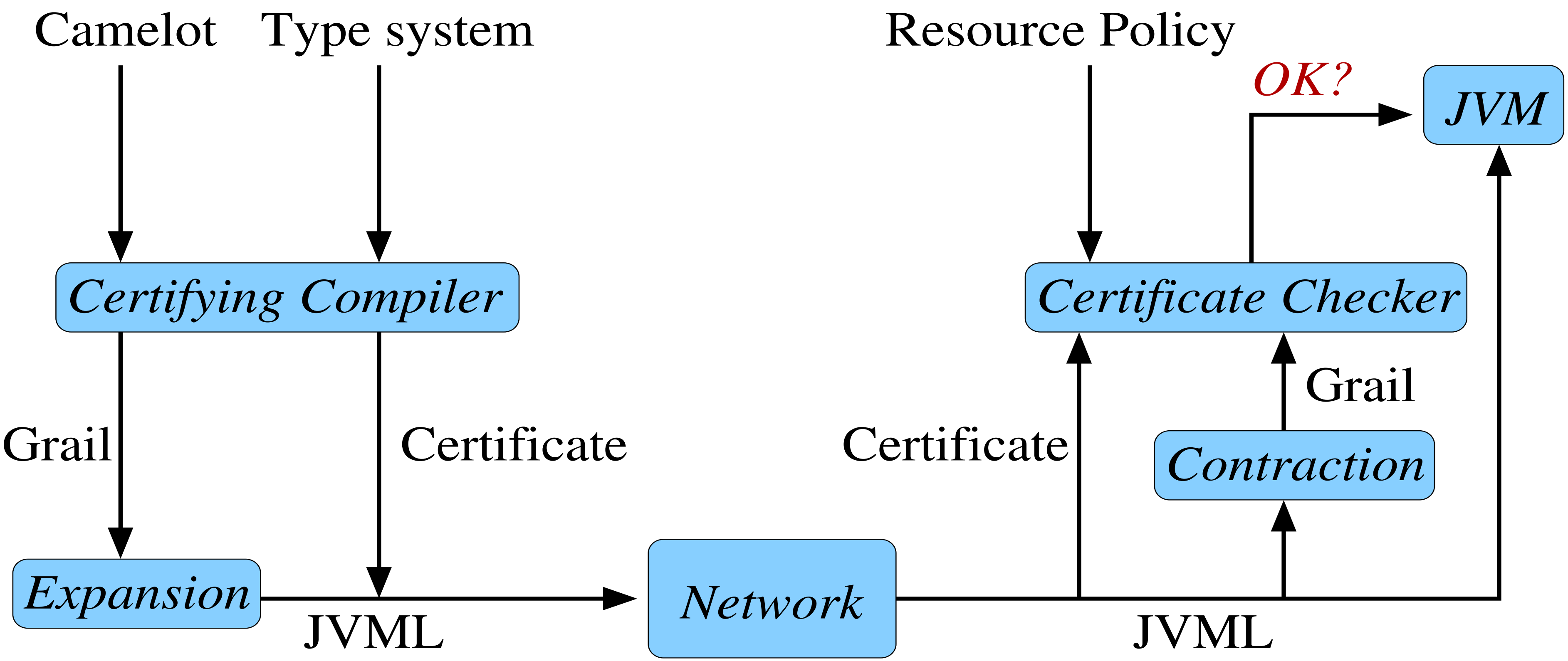
MRG: PCC infrastructure for resource-related properties

- 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

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 JVML.
- Costs are calculated using a **annotated operational semantics** for Grail, reflecting the expansion into JVML.
- **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



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 (Princeton): operational model (processor) formalised in higher-order logic built on top of theorem prover (e.g. Twelf/HOL).
- “Yale-style” PCC . . .

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 for high level language) in theorem prover
- soundness of the heap logics with respect to the operational model is obtained from the soundness of the base logic.
- the type systems infers invariants (in our case: method specifications) for the low-level code based on the strategy used for compiling high-level programs.
- the proof rules are set up in such a way that methods can be proved in a largely syntax-directed way, with side conditions that are of low complexity.

Source language: 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)
- PCC-certificate: encoding of the result of the type inference in a program logic, bundled with program for transmission
- 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)
- O'Camelot: object-oriented extension (see SML.net).

Mobile code: Grail 1/2

- A subset of Java bytecode. Combine OO-aspects of bytecode (fields, methods) with (impure) low-level functional language
- View as a functional intermediate language: first-order functions; no nesting; all free variables in parameters; applications only to values.
- Imperative view: JVML or easily convertible into various virtual machines formats: registers = variables, jumps = tail-calls
- Theorem: the two coincide under mild syntactical restrictions (Leroy's bytecode condition)
- This makes conversion Grail/JVML reversible
- A Grail program is a list of *methods* each containing a list of tail-recursive *functions*.

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

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* $\mathcal{R}$, with families of operations for each of the syntactic constructs of Grail:

- JVM case: R consists of quadruples:

$$r = (\textit{clock}, \quad \textit{callc}, \quad \textit{invkc}, \quad \textit{invkdpth})$$

- Stack usage is approximated; heap usage calculated as the difference $\textit{size}(h') - \textit{size}(h)$.
- Resource algebras usefully generalise to other resource/security policies
  - *parameter limit flags* set by parameter limit policies; here simply $R = \{\text{true}, \text{false}\}$.
  - *traces of method invocation sequences*, so e.g. $R = \{m^*\}$ where m ranges over method names.
  - *read-write effects on heap locations*, where $R = \{\langle \text{Rd}, \text{Or}, \text{RdWr} \rangle\}$ for $\text{Rd}, \text{Wr}, \text{RdWr} \subseteq \textit{Locations}$.
  - Others: live variables, complete traces of heaps during execution, . . .

Demo: what you're going to see

- Producer side:
- High level source code: `Insort.cmlt`. Certifying compiler emits:
  1. Bytecode: `Insort.class, Insort$$_dia.class`
  2. Inference of heap consumption: `Insort.lfd`
  3. Isabelle theory certificate containing above specs: `InsortCertificate.thy`
- Consumer side. De-assembler emits
  1. Isabelle representation of mobile code: `Insort.thy`
  2. Isabelle statement of resource predicate and related lemmas `Insort_Consumer1.thy, Insort_Consumer2.thy`
  3. Isabelle tactic to reconstruct proof: `Insort_TACTIC.thy`

Program logic 1/2

- 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 $G \rhd e : P$
  - e is a Grail expression;
  - G is a set of assumptions context for recursive methods and functions;
  - P is an assertion, i.e. a predicate in the meta-logic over semantic values

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

    relating the environment, initial and final heaps, the result and the resource value.
- No auxiliary variables (usage of pre-heap inspired by hooked variables in VDM)
- Judgements interpreted as partial "correctness" statements. Termination orthogonal.

$$\frac{}{\begin{array}{r}G \rhd x.t : \lambda\, E\, h\, h'\, v\, p.\, \exists\, l.\;\; E\langle x\rangle = \mathit{Ref}\ l \;\wedge\; h' = h \;\wedge \\ v = h'(l).t \;\wedge\; p = \mathcal{R}^{\texttt{getf}}(x, t)\end{array}} \quad (\text{VGETF})$$

Program logic 2/2: example specification

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

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

Lemma: *insSpec* $\wedge$ *sortSpec* $\longrightarrow\ \rhd$ List.$sort([xs])$: *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

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**$(\mathtt{l})$ will not call **new** when evaluated in a heap $\mathtt{h}$ where

  - $\mathtt{l}$ points in $\mathtt{h}$ to a linear list of some length $\mathtt{n}$
  - the freelist which forms a part of $\mathtt{h}$ is well-formed
  - the freelist does not overlap with $\mathtt{l}$
  - the freelist has length not less than $s(\mathtt{n})$

- How does this work?

  - Annotate types with freelist annotations for each constructor: **L**$(\mathtt{k})$
  - Judgements $\Gamma, \mathtt{n} \vdash e : \mathsf{T}, \mathtt{m}$ 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;
  - compile-time analysis is communicated as method-level specifications (invariants)

  item
- Fixed assertion format $[\![U, n, [\Gamma] \blacktriangleright T, m]\!]$

  $n, m \in \mathbb{N}$ represent the numerical results from the analysis. In the interpretation these numbers will relate to the initial and final length of the freelist, respectively.

  Γ is the typing context, a partial map from program variables to extended types.

  U (a finite set of program variables) is used to enforce the linear typing discipline.

  T indicates the type of an expression e that satisfies the assertion.

Proof rules

- Camelot extended typing

$$\begin{aligned} \text{List.ins} &: 1, \mathbf{I} \times \mathbf{L}(0) \to \mathbf{L}(0), 0 \\ \text{List.sort} &: 0, \mathbf{L}(0) \to \mathbf{L}(0), 0 \end{aligned}$$

- Derived assertions:

$$\begin{aligned} \text{List.ins} &: [\![\{a, l\}, 1, [a : \mathbf{I}, l : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \\ \text{List.sort} &: [\![\{l\}, 0, [l : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0]\!] \end{aligned}$$

- LFD rule (Let):

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

- Note linearity condition for eliminating deallocation of live cells
- Proof rule (Let), provided $U_1 \cap (U_2 \setminus \{x\}) = \emptyset$:

$$\frac{G \rhd e_1 : [\![U_1, n, [\Gamma] \blacktriangleright S, k]\!] \qquad G \rhd e_2 : [\![U_2, k, [\Gamma, x : S] \blacktriangleright T, m]\!]}{G \rhd \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : [\![U_1 \cup (U_2 \setminus \{x\}), n, [\Gamma] \blacktriangleright T, m]\!]} \dagger$$

- Atomic rules for [non] destructive match-statements and for invocations of **make**
- Only the verification of the wrapper (uniform for all programs) needs to unfold the

Automated verification

- 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, . . .)
  - No effort whatsoever on efficiency/proof sizes/negotiation. . .
  - On-going generalization to algebraic data-type and usage aspects.

Discussion

Future work:

- Engineer 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
- Make them compositional
- Mobius: play this game with Java as source language

Conclusion:

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