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

Camelot  Type system

Certifying Compiler

Grail

Expansion

JVML

Certificate

Network

Resource Policy

Certificate Checker

Grail

Contraction

JVML

OK?
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:

  ```camelot
  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.

\[
\begin{align*}
\text{expr} & ::= \text{null} \mid i \mid x \mid \text{prim}\ p \mid x \\
& \quad \mid \text{new}\ c\ \left[ t_i := x_i \right] \\
& \quad \mid x.t \mid x.t := x \\
& \quad \mid \text{let}\ x = e\ \text{in}\ e \mid e;e \\
& \quad \mid \text{if}\ x\ \text{then}\ e\ \text{else}\ e \\
& \quad \mid \text{call}\ f \mid c.m(a) \\
\text{args} & ::= x \mid \text{null} \mid i
\end{align*}
\]
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: \( \mathcal{R} \) consists of quadruples:

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

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

- Resource algebras usefully generalise to other resource/security policies
  - 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 \( \text{Rd}, \text{Wr}, \text{RdWr} \subseteq \text{Locations} \).
  - 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 $G \triangleright e : P$
  - $e$ 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

$$P[E, h, h', \nu, r]$$

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.

\[ G \triangleright x.t : \lambda E h h' \nu p. \exists l. E(x) = \text{Ref } l \land h' = h \land \nu = h'(l).t \land p = \mathcal{R}^{getf}(x, t) \]
Program logic 2/2: example specification

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

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

Lemma: \( insSpec \land sortSpec \rightarrow \triangleright \text{List.sort } ([xs]) : SPEC \text{ List sort } [xs] \)

- \( h, r \models_X n \) defined inductively, introduces case-splits during verification
- Proof rules contain existentials over intermediate heaps and instrumentations
- \( \rightarrow \) 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 \texttt{start}: \texttt{string list -> unit}
  Output: a \textit{linear function} \(s\) such that \texttt{start}(l) will not call \texttt{new} when evaluated in a heap \(h\) where
    - \(l\) 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 \(l\)
    - the freelist has length not less than \(s(n)\)

- How does this work?
  - Annotate types with freelist annotations for each constructor: \(L(k)\)
  - Judgements \(\Gamma, \eta \vdash e : T, m\) 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;
  - compile-time analysis is communicated as method-level specifications (invariants)
- Fixed assertion format $\llbracket U, n, [\Gamma] \Rightarrow T, m \rrbracket$
  $n, m \in \mathbb{N}$ represent the numerical results from the analysis. In the interpretation these numbers will relate to the initial and final length of the freelist, respectively.
  $\Gamma$ 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

List.ins : \( \mathbb{1}, \mathbb{1} \times \mathbb{L}(0) \rightarrow \mathbb{L}(0), 0 \)

List.sort : \( \mathbb{0}, \mathbb{L}(0) \rightarrow \mathbb{L}(0), 0 \)

- Derived assertions:

List.ins : \( \llbracket \{ a, l \}, 1, [ a : \mathbb{1}, l : \mathbb{L}(0) ] \rrbracket \rightarrow \mathbb{L}(0), 0 \)

List.sort : \( \llbracket \{ l \}, 0, [ l : \mathbb{L}(0) ] \rrbracket \rightarrow \mathbb{L}(0), 0 \)

- LFD rule (Let):

\[
\Gamma_1, n \vdash e_1 : A, k \quad \Gamma_2, x : A, k \vdash e_2 : B, m
\]

\[
\Gamma_1 \Gamma_2, n \vdash \text{let } x = e_1 \text{ in } e_2 : B, m
\]

- Note linearity condition for eliminating deallocation of live cells

- Proof rule (Let), provided \( \mathcal{U}_1 \cap (\mathcal{U}_2 \setminus \{ x \}) = \emptyset \):

\[
G \triangleright e_1 : \llbracket \mathcal{U}_1, n, [\Gamma] \triangleright S, k \rrbracket \quad G \triangleright e_2 : \llbracket \mathcal{U}_2, k, [\Gamma, x : S] \triangleright T, m \rrbracket
\]

\[
G \triangleright \text{let } x = e_1 \text{ in } e_2 : \llbracket \mathcal{U}_1 \cup (\mathcal{U}_2 \setminus \{ x \}), n, [\Gamma] \triangleright T, m \rrbracket
\]

- Atomic rules for [non] destructive match-statements and for invocations of \texttt{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 $\rightarrow$ 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: