Towards Certification of Resource Consumption

Alberto Momigliano and Lennart Beringer
Laboratory for Foundations of Computer Science
University of Edinburgh

Work carried out in the

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

Reasoning Seminar, November, 5th, 2004
MRG: PCC infrastructure for resource-related properties

- 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
- Approach (certifying compilation): translation from user language into machine language derives independently verifiable certificates
- MRG complements security usages of PCC (memory safety, . . .)

This talk: brief overview, short demo, “how does it work”? 
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:

  ```ml
  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

- Memory consumption inferred from program annotations using a type system
- Result: \texttt{ins} consumes one memory cell, independent from actual input), \texttt{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 (no type inference, just proof checking in Isabelle) before executing program
Demo: what you are going to see

- Example: insertion sort
- Valid method specification: in-place-property
- Execution of MRGjava: compilation to JVM, certificate checking succeeds, execution of program
- Invalid specification: claims that one memory cell is gained
- Execution of MRGjava: certificate checking fails
Example

List reversal (obtained from Camelot code, pretty printed)

method LST.rev(l, acc) = if l.TAG = 0 then return acc
else h = l.HD; t = l.TL;
   l.TAG := 1; l.HD := h;
   l.TL := acc; return LST.rev(t,l)

Specification (no functional correctness, just resources):

\[
ST \ LST.\ rev \ z = \lambda \ E \ h \ h' \ \nu \ p. \ \forall \ n \ a \ X \ m \ b \ Y. \\
\left( \begin{array}{c}
(E \ z = [Ref \ a, Ref \ b] \ \land \ h, a \models_X n \ \land \ h, b \models_Y m \ \land \ X \cap Y = \emptyset) \\
\rightarrow |dom(h)| = |dom(h')| \ \land \ p = \langle (29n + 13) \ 0 \ (n + 1) \ (n + 1) \rangle
\end{array} \right)
\]

Verification doable but cumbersome (\exists\text{-instantiations, case splits,\ldots})
Derived logics: linear heap consumption

- Idea: Develop specialised logics whose proof rules are related to type systems at high and intermediate language level
- Exploit structure of Camelot compilation and analysis
- Certificate generation largely done by type inference
- LRPP: Interpret judgements of Hofmann/Jost type system by representing the soundness statement in the program logic, i.e. all specifications are of the same restricted form
- “Certificate”: method specifications, verification of bodies fully automated, syntax-directed, with simple side conditions
- Examples: insertion sort, heap sort etc
Future/current work

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

- Presented expressive program logic for low-level language
- Single assertion style, cut rules for mutual recursion and parameter adaptation
- 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:
  - Classic: extract stand-alone proof checker
  - Foundational: unfold to core logic or operational semantics
  $\rightsquigarrow$ Proof negotiation
How does it work?

Existing approaches:

- Classical PCC: trusted special-purpose proof system for proving light-weight properties of machine code (memory safety)
- Foundational PCC: operational model (processor) formalised in general-purpose logic, special-purpose logic derived from this model, again in general-purpose theorem prover

MRG:

- Formalise *instrumented* operational semantics
- 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: ANF-style + further syntactic restrictions
- Imperative view: easily convertible into various VM formats
- registers = variables, jumps = tail-calls
- Coincidence between functional and imperative views makes conversion reversible
- Emitted bytecode is highly structured (Leroy’s conditions)
Formalisation of Grail

- Named syntax (no HOAS)

\[
\text{datatype expr =}
\begin{align*}
&\text{Int } \text{int} \\
&\mid \text{Primop } (\text{int }\Rightarrow \text{int}) \text{name name} \\
&\mid \text{New } \text{cname } (\text{fldname name}) \text{list} \\
&\mid \text{GetF } \text{name fldname} \\
&\mid \text{PutF } \text{name fldname name} \\
&\mid \text{InvokeStatic } \text{cname mname ARGTYPE} \\
&\mid \text{Let } \text{name expr expr} \\
&\mid \text{Ifg } \text{name expr expr} \\
&\mid \text{Call } \text{funame}
\end{align*}
\]

- Program encoded using global tables (functions and methods)

- Impure functional semantics based on (finite) maps:

\[
\begin{align*}
\text{env} &= \text{name }\Rightarrow \text{val} \\
\text{heap} &= \text{locn }\Rightarrow f \text{ cname} \\
&\quad \text{fldname }\Rightarrow \text{locn }\Rightarrow \text{val} \\
&\quad \text{cname }\Rightarrow \text{fldname }\Rightarrow \text{ref}
\end{align*}
\]
Grail: resource-instrumented operational semantics

Based on (impure) functional view:

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

Resource component \( P \) models costs, and can be instantiated to instruction counters corresponding to executed JVM instructions, invocation depth, satisfaction of parameter value policies and other observations

\[ E \vdash h, e_1 \Downarrow (h_1, w, p) \quad w \neq \bot \quad E\langle x := w \rangle \vdash h_1, e_2 \Downarrow (h_2, v, q) \]

\[(\text{LET})\]

\[ E \vdash h, \text{let } x = e_1 \text{ in } e_2 \Downarrow (h_2, v, P^{\text{let}}(x, p, q)) \]

\[ E\langle x \rangle = \text{Ref } l \]

\[(\text{GETF})\]

\[ E \vdash h, x.t \Downarrow (h, h(l).t, P^{\text{getf}}(x, t)) \]

where \( P^{\text{getf}}(x, t) = \langle 2 0 0 0 \rangle \) or...
Program logic I

- General reappraisal of program (Hoare) logics: embeddings in theorem prover (Kleymann, Nipkow), Separation logics (Reynolds, O'Hearn), Java verification (Jacobs, de Boer, vonOheimb)
- Embedding a la Nipkow: 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!
- Specifications $\mathcal{A}$ are predicates over semantic components evaluation environment (local variables), initial & final heap, result value, and resource component
- No auxiliary variables (usage of post-heap inspired by hooked variables in VDM)
- Judgements interpreted as partial “correctness” statements: validity $\models e : \mathcal{A}$ defined as

\[
\forall E \ h \ h' \ v \ p. \ (E \vdash \ h, e \Downarrow (h', v, p) \longrightarrow \mathcal{A} \ E \ h \ h' \ v \ p).
\]

- Termination considered orthogonal
Program logic III: proof rules

\[ \Gamma \triangleright e_1 : A_1 \quad \Gamma \triangleright e_2 : A_2 \]

\[ \Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \lambda E h h' \nu p. \exists p_1 p_2 h_1 w. (A_1 E h h_1 w p_1) \land w \neq \bot \land \]

\[ (A_2 (E(x := w)) h_1 h' \nu p_2) \land \]

\[ p = P^{\text{let}}(x, p, q) \]

(VLET)

\[ \Gamma \triangleright x.t : \lambda E h h' \nu p. \exists l. E(x) = \text{Ref } l \land h' = h \land \]

\[ \nu = h'(l).t \land p = P^{\text{getf}}(x, t) \]

(VGETF)

- Structural rules: context lookup and rule of consequence
- Admissable rules (derived in Isabelle): cut
- Context \( \Gamma \) stores recursive assumptions. \( \rightsquigarrow \) proof system suffices for mutual recursion and parameter adaptation of method calls
Program logic IV: soundness & completeness

Follows earlier work by Kleymann, Nipkow, and Hofmann.

- Soundness proven as usual, by relativised validity and induction on height of derivations
- Shallow embedding: avoids definition of language and logic of assertions
- “Relative” completeness: in rule of consequence, the implication only needs to *hold* rather than being *derivable*
- Implementation in theorem prover using shallow embedding: use the meta-logical implication. $\rightarrow$ incompleteness of meta-logic (HOL) is inherited by program logic
- Completeness proven by defining strongest specifications, a specification table $\hat{ST}$ associating to each function call / method invocation its strongest specification, proving that the corresponding context is *good* w.r.t. $\hat{ST}$, and applying (a variant of) the cut rule and MUTREC.
Program logic V: example specification (insertion sort)

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

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

Lemma: \(\text{insSpec} \land \text{sortSpec} \to List \diamond \text{sort}([xs]) : MS \text{ List sort } [xs]\)

- \(h, r \models_X n\) defined inductively, introduces case-splits during verification
- proof rules contain existentials over intermediate heaps and instrumentations
- \(\leadsto\) automatic proof search impractical even after applying all proof rules (VCG):50-100 Isar-commands
- \(\not\to\) certificate generation by compiler difficult
- Certificate Generation: exploit program structure and compiler analysis by proving properties that are more closely related to the type system
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 \texttt{start: 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: \texttt{iTree(n, m)}
  
  - Judgements \(\Gamma, n \vdash e : T, m\) include information about \textit{initial} and \textit{final} size of freelist
  
  - Express final size of freelist as function of the size of the output
  
  - Complement this type system with an arbitrary 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...
  - provided that results of the compile-time analysis are communicated as method-level
    specifications (invariants)
Proof rules

• Chose linearity condition for eliminating deallocation of live cells
  \[ \leadsto \text{proof rules are expressed at a level where program variables occur (affinely) linear} \]

• Linear context implemented in two components

• Example rule (Let)

\[
\frac{G \triangleright e_1 : [U_1, n, [\Gamma] \gg S, k]}{G \triangleright \text{let } x = e_1 \text{ in } e_2 : [U_1 \cup (U_2 \setminus \{x\}), n, [\Gamma] \gg T, m]} \quad U_1 \cap (U_2 \setminus \{x\}) = \emptyset
\]

• Atomic rules for (destructive and non-destructive) match-statements and for invocations of make

• Example rule (ListMatchD)

\[
\frac{\Gamma(x) = L(k) \quad G \triangleright e : [U, n + k + 1, [\Gamma, h : l, t : L(k)] \gg T, m] \quad x \notin U \cup \{h, t\}}{G \triangleright \text{let } h = x.\text{HD in let } t = x.\text{TL in } D \odot \text{free}(x) ; e : [(U \setminus \{h, t\}) \cup \{x\}, n, [\Gamma] \gg T, m]}
\]

• 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 `prove`

Consumer side:

- Tactic `prove` 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 tested on 11 methods (append, flatten, insertion sort & heap sort)
  - Runtime (inside Isabelle environment) between 2secs and 30secs