From Bytecode Logic to Certificate Generation for Grail

Alberto Momigliano
Laboratory for Foundations of Computer Science
University of Edinburgh

Joint work with D. Aspinall, L. Beringer, H-W. Loidl, M. Hofmann and O. Shkaravska

Global Computing, 11th March 2004
Overview

- Brief review of the Grail language and its (functional) operational semantics

- A new program (bytecode) logic for reasoning about resource consumption, formalised in Isabelle

- A specialised logic for automatic certificate generation, linking the program logic to resource (LFD) inference in Camelot, effectively closing the gap between high-level static analysis and general theorem proving
Guaranteed Resource Aware Intermediate Language

- Grail is a key component of the MRG platform
- Abstract representation of virtual machine languages (JVM)
  - 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 guaranteed code
  - Executable

- Grail mediates between all of these roles by having two distinct semantic interpretations, one functional and one imperative
Imperative Grail

- Grail 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.
Functional Grail

• Grail also has a functional semantics (with side-effects):
  – Strong static typing
  – Call-by-value first-order functions
  – Local function declarations
  – Mutual recursion
  – Lexical scoping of variables and parameters

• This simple functional language is the target for the Camelot high-level language compiler.
What makes it work

• The coincidence of the semantics holds because we place tight constraints on well-formed program (Grail normal forms)
  – No nesting: only one level of local functions
  – tail calls only
  – Functions must include all free variables as parameters
  – Functions are only applied to variables, which must syntactically coincide with the parameter names

• Imperative Grail is similarly well-behaved: the operand stack is empty at all jumps and branches. This enables reverse JVM translation
Grail: Operational semantics

- Big step operational semantics with cost model \( E \vdash h, e \Downarrow (h', v, p) \) relating expression \( e \), environment \( E \), (pre-)heap \( h \), result \( v \), (post-)heap \( h' \) and cost component

\[
p = \langle \text{clock} \quad \text{callc} \quad \text{invkc} \quad \text{invkdepth} \rangle.
\]

- Instruction counter models \# of JVM instructions in imperative unfolding, size of heap inferred inferred

- Program represented by global tables for function and method declarations (\( MT \))

- Method frames modelled implicitly by creating new environments (function \textit{newframe})
Operational semantics: Sample rules

\[ E \vdash h, \text{var } x \downarrow (h, E\langle x \rangle, \langle 1 \ 0 \ 0 \ 0 \rangle) \]  
\hspace{10pt} (\text{var})

\[ E\langle x \rangle = \text{Ref } l \]
\[ E \vdash h, x.t := y \downarrow (h[l.t \mapsto E\langle y \rangle], \bot, \langle 3 \ 0 \ 0 \ 0 \rangle) \]  
\hspace{10pt} (\text{putf})

\[ E \vdash h, e_1 \downarrow (h_1, w, p) \quad w \neq \bot \quad E\langle x := w \rangle \vdash h_1, e_2 \downarrow (h_2, v, q) \]
\[ E \vdash h, \text{let } x = e_1 \ \text{in } e_2 \downarrow (h_2, v, \langle 1 \ 0 \ 0 \ 0 \rangle - p - q) \]  
\hspace{10pt} (\text{let})

\[ (\text{newframe} \ldots \bar{a} \ E) \vdash h, \text{MT } c \ m \downarrow (h_1, v, p) \]
\[ E \vdash h, c \circ m(\bar{a}) \downarrow (h_1, v, \langle (2 \uparrow | \bar{a} |) \ 0 \ 1 \ 1 \rangle \oplus p) \]  
\hspace{10pt} (\text{sinv})
We use program logics to generate verification conditions.

We adapt VDM style to functional setting: specifications \( A \) are predicated in HOL over \( E \times H \times H \times V \times R \), no syntactic separation into pre- and post-conditions.

Logic for partial correctness: judgement \( \models e : A \) means “whenever \( E \vdash h, e \Downarrow (v, h', p) \) then \( A \ E \ h \ h' \ v \ p \) holds”

More flexible than hardwired VCGen. Crucial: infrastructure has a formalized soundness and completeness proof.

Logic designed as the basis for concrete program verification.

Termination, more that total corr’ness, orthogonal (Amadio).
• Sample rule format: parameterless static method invocation

\[ \Gamma, c \diamond m() : A \triangleright e : A^+ \]
\[ \Gamma \triangleright c \diamond m() : A \]

where \( e \) is the body of \( c \diamond m() \), \( A^+ \) is

\[ \lambda E h h' v p. \phi(E, h, h', v, A^+) \]

and \( p^+ \) is updated cost component

• Context \( \Gamma \) collects hypothetical judgements for recursion: we verify the body under the assumption that further invocations satisfy the specification
Program logic rules

• Derivation system \( \Gamma \triangleright e : A \)

\[
\Gamma \triangleright \text{var } x : \lambda E h h' v p. h' = h \land v = E\langle x \rangle \land p = \langle 1 0 0 0 \rangle
\]

\[
\Gamma \triangleright x. t := y : \lambda E h h' v p. \exists l. E\langle x \rangle = \text{Ref } l \land p = \langle 3 0 0 0 \rangle \land h' = h[l.l \mapsto E\langle y \rangle] \land v = \bot
\]

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

\[
\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \lambda E h h' v p. \exists p_1 p_2 h_1 w. \ (A_1 E h h_1 w p_1) \land w \neq \bot \land (A_2 (E\langle x := w \rangle) h_1 h' v p_2) \land p = \langle 1 0 0 0 \rangle \oplus (p_1 \sim p_2)
\]

\[
\Gamma, c \diamond m(\vec{a}) : A \triangleright MT c \ m : \lambda E h h' v p. \forall E'. E = (\text{newframe null pars } c \ m \ \vec{a} \ E') \quad \rightarrow A E' h h' v \langle (2+ | \vec{a} |) 0 1 1 \rangle \oplus p
\]

\[
\Gamma \triangleright c \diamond m(\vec{a}) : A
\]
• Technical contribution: new rules for mutually recursive methods and for parameter adaptation in method invocations, avoiding syntactical/semantical substitutions:

\[
\frac{(\Gamma, e : A) \text{ goodContext}}{\triangleright e : A} \text{ Mutrec}
\]

\[
\frac{(\Gamma, c \diamond m(\bar{a}) : MS \; c \; m \; \bar{a}) \text{ goodContext}}{\triangleright (c \diamond m(\bar{b}) : MS \; c \; m \; \bar{b})} \text{ Adapt}
\]

• Proven via admissible Cut rule, no extra derivation system

• Global specification table \( MS \), \( \text{goodContext} \) relates entries in \( MS \) to the method bodies
**Example: Insertion sort**

```plaintext
method static public List ins (int a, List l) = ...Make(..., ..., ...)
method static public List sort (List l) =
    let fun f(List l) =
        if l = null then null
        else let val h = l.HD
             val t = l.TL
             val () = D.free (l)
             val l = List.sort (t)
         in List.ins (h, l) end
    in f(l) end
```

```
insSpec ≡ MS List ins [a₁, a₂] =
    λ E h h' v p . ∀ i r n X .
    ( E⟨a₁⟩ = i ∧ E⟨a₂⟩ = Ref r ∧ h, r ≡ₜ n
    → |dom (h)| + 1 = |dom (h')| ∧
    p ≤ ⟨(An + B) (Cn + D) (En + F) (Gn + H)⟩)
```

```
sortSpec ≡ MS List sort [a] =
    λ E h h' v p . ∀ i r n X .
    ( E⟨a⟩ = Ref r ∧ h, r ≡ₜ n → |dom (h)| = |dom (h')| ∧ p ≤ ...)
```

**Lemma:** insSpec ∧ sortSpec → ▷ List ◁ sort([xs]) : MS List sort [xs]
Discussion of core logic

- Expressive logic for correctness and resource consumption
- Less suited for immediate program verification: not fully automatic (case-splits, $\exists$-instantiation,...), verification conditions large and complex
- Continue abstraction: loop unfolding in op. semantics $\rightarrow$ invariants in general program logics $\rightarrow$ specific logic for resource properties
- Aim: exploit structure of Camelot compilation (freelist) and program analysis

\begin{align*}
\text{List.ins} & : 1, \text{int} \times \text{list}(0) \rightarrow \text{list}(0), 0 \\
\text{List.sort} & : 0, \text{list}(0) \rightarrow \text{list}(0), 0
\end{align*}
LFD-assertions

- Translation of Hofmann-Jost type system to Grail, types interpreted as relating initial to final freelist

- Fixed assertion format \([U, n, [\Delta] \rightarrow T, m]\)

  \[
  \begin{align*}
  \text{List.ins} : & \quad [\{a, l\}, 1, [a : \text{int}, l : \text{list}(0)] \rightarrow \text{list}(0), 0] \\
  \text{List.sort} : & \quad [\{l\}, 0, [l : \text{list}(0)] \rightarrow \text{list}(0), 0]
  \end{align*}
  \]

- LFD types express space requirements for datatype constructors, numbers \(n, m\) refer to the freelist length

- Semantic definition by expansion into core bytecode logic, derived proof rules using linear affine context management

- Early experience: dramatic reduction of VC complexity
Semantic interpretation of \([U, n, [\Delta] \triangleright T, m]\)

\[
[U, n, [\Delta] \triangleright T, m] \equiv \\
\lambda E \ h \ h' \ v \ p. \\
\forall F \ N. \ (\text{regionsExist}(U, \Delta, h, E) \land \text{regionsDistinct}(U, \Delta, h, E) \land \\
\text{freelist}(h, F, N) \land \text{distinctFrom}(U, \Delta, h, E, F)) \rightarrow \\
(\exists R \ S \ M \ G. \ v, h' \models_T R, S \land \text{freelist}(h', G, M) \land R \cap G = \emptyset \land \\
\text{Bounded}((R \cup G), F, U, \Delta, h, E) \land \text{modified}(F, U, \Delta, h, E, h') \land \\
\text{sizeRestricted}(n, N, m, S, M, U, \Delta, h, E) \land \text{dom} \ h = \text{dom} \ h')
\]

**IF** the variables in \(U\) point to disjoint regions and the initial freelist is of length \(N\), and disjoint from the variables **THEN**

- then there are numbers \(M\) and \(S\), and regions \(R\) and \(G\) such that the result \(v\) is of size \(S\) and disjoint from freelist
- the final freelist is of length \(M\), as claimed by analysis and bounded by initial freelist and \(U\)-regions
- variables outside \(U\) remain unchanged, neither new objects are allocated.
Proof system

- Proof system with linear inequalities and linear affine type system \((U, \Delta)\) that guarantees benign sharing

\[
\Delta(x) = T \quad n \leq m
\]

\[
\frac{\Gamma \triangleright \text{var } x : \llbracket \{x\}, m, [\Delta] \triangleright T, n]}{(\text{Var})}
\]

\[
\Gamma \triangleright e_1 : \llbracket U_1, n, [\Delta] \triangleright T_1, m] \quad \Gamma \triangleright e_2 : \llbracket U_2, m, [\Delta, x : T_1] \triangleright T_2, k]
\]

\[
U_1 \cap (U_2 \setminus \{x\}) = \emptyset \quad T_1 = \text{list(\_)}
\]

\[
\frac{\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \llbracket U_1 \cup (U_2 \setminus \{x\}), n, [\Delta] \triangleright T_2, k]}{(\text{Let})}
\]

\[
\Delta(x) = \text{list}(k) \quad l = n + k \quad \Gamma \triangleright e : \llbracket U, l, [\Delta, t : \text{list}(k)] \triangleright T, m] \quad x \notin U \setminus \{t\}
\]

\[
\frac{\Gamma \triangleright \text{let } t = x.TL \text{ in } e : \llbracket (U \setminus \{t\}) \cup \{x\}, n, [\Delta] \triangleright T, m]}{(\text{LetTL})}
\]

- Linearity relaxed in rules for compiled match-expressions

- VCG: infer the usage-sets \(U\) (essentially type checking) and verify that inequalities hold
Conclusion

- We have presented a program logic for reasoning about resource consumption in Grail
- We have specialised it to exploit program structure and compiler analysis: most effort done once (in soundness proofs), application straight-forward
- “Classic PCC”: independence of derived logic from Isabelle (no higher-order predicates, certifying constraint logic programming)
- “Foundational PCC”: can unfold back to core logic and operational semantics if desired
- Future work: generalisation to arbitrary Camelot datatypes needed and more liberal sharing disciplines