

# 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

$$p = \langle \text{clock} \quad \text{callc} \quad \text{invkc} \quad \text{invkdpth} \rangle.$$

- Instruction counter models # of JVM instructions in imperative unfolding, size of heap inferred
- Program represented by global tables for function and method declarations ( $MT$ )
- Method frames modelled implicitly by creating new environments (function  $\text{newframe}$ )

## Operational semantics: Sample rules

---

$$\frac{}{E \vdash h, \text{var } x \Downarrow (h, E\langle x \rangle, \langle 1 \ 0 \ 0 \ 0 \rangle)} \quad (\text{var})$$

$$\frac{E\langle x \rangle = \text{Ref } l}{E \vdash h, x.t := y \Downarrow (h[l.t \mapsto E\langle y \rangle], \perp, \langle 3 \ 0 \ 0 \ 0 \rangle)} \quad (\text{putf})$$

$$\frac{E \vdash h, e_1 \Downarrow (h_1, w, p) \quad w \neq \perp \quad E\langle x := w \rangle \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, \text{let } x = e_1 \text{ in } e_2 \Downarrow (h_2, v, \langle 1 \ 0 \ 0 \ 0 \rangle \smile p \smile q)} \quad (\text{let})$$

$$\frac{(\text{newframe } \dots \ \bar{a} \ E) \vdash h, MT \ c \ m \Downarrow (h_1, v, p)}{E \vdash h, c \diamond m(\bar{a}) \Downarrow (h_1, v, \langle (2+ |\bar{a}|) \ 0 \ 1 \ 1 \rangle \oplus p)} \quad (\text{sinv})$$

- We use program logics to generate verification conditions
- We adapt VDM style to functional setting: specifications  $A$  are predicated in HOL over  $\mathcal{E} \times \mathcal{H} \times \mathcal{H} \times \mathcal{V} \times \mathcal{R}$ , 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 than total correctness, orthogonal (Amadio)

- Sample rule format: parameterless static method invocation

$$\frac{\Gamma, c \diamond m() : A \triangleright e : A^+}{\Gamma \triangleright c \diamond m() : A}$$

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

- Derivation system  $\Gamma \triangleright e : A$

$$\frac{}{\Gamma \triangleright \text{var } x : \lambda E h h' v p. h' = h \wedge v = E\langle x \rangle \wedge p = \langle 1 \ 0 \ 0 \ 0 \rangle}$$

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

$$\Gamma \triangleright e_1 : A_1 \quad \Gamma \triangleright e_2 : A_2$$

$$\frac{}{\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \lambda E h h' v p. \exists p_1 p_2 h_1 w. (A_1 E h h_1 w p_1) \wedge w \neq \perp \wedge (A_2 (E\langle x := w \rangle) h_1 h' v p_2) \wedge p = \langle 1 \ 0 \ 0 \ 0 \rangle \oplus (p_1 \smile p_2)}$$

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

- 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

---

```

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

```

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

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

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

## 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

List.ins :  $1, \text{int} \times \text{list}(0) \rightarrow \text{list}(0), 0$   
List.sort :  $0, \text{list}(0) \rightarrow \text{list}(0), 0$

## LFD-assertions

---

- Translation of Hofmann-Jost type system to Grail, types interpreted as relating initial to final freelist
- Fixed assertion format  $\llbracket U, n, [\Delta] \blacktriangleright T, m \rrbracket$ 
  - List.ins :  $\llbracket \{a, l\}, 1, [a : \text{int}, l : \text{list}(0)] \blacktriangleright \text{list}(0), 0 \rrbracket$
  - List.sort :  $\llbracket \{l\}, 0, [l : \text{list}(0)] \blacktriangleright \text{list}(0), 0 \rrbracket$
- 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 $\llbracket U, n, [\Delta] \blacktriangleright T, m \rrbracket$

---

$$\begin{aligned} \llbracket U, n, [\Delta] \blacktriangleright T, m \rrbracket \equiv & \\ \lambda E h h' v p. & \\ \forall F N. & \left( \text{regionsExist}(U, \Delta, h, E) \wedge \text{regionsDistinct}(U, \Delta, h, E) \wedge \right. \\ & \left. \text{freelist}(h, F, N) \wedge \text{distinctFrom}(U, \Delta, h, E, F) \right) \longrightarrow \\ & \left( \exists R S M G. v, h' \models_T R, S \wedge \text{freelist}(h', G, M) \wedge R \cap G = \emptyset \wedge \right. \\ & \left. \text{Bounded}((R \cup G), F, U, \Delta, h, E) \wedge \text{modified}(F, U, \Delta, h, E, h') \wedge \right. \\ & \left. \text{sizeRestricted}(n, N, m, S, M, U, \Delta, h, E) \wedge \text{dom } h = \text{dom } h' \right) \end{aligned}$$

**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

$$\frac{\Delta(x) = T \quad n \leq m}{\Gamma \triangleright \text{var } x : [\{x\}, m, [\Delta] \blacktriangleright T, n]} \quad (\text{Var})$$

$$\frac{\begin{array}{l} \Gamma \triangleright e_1 : [U_1, n, [\Delta] \blacktriangleright T_1, m] \\ U_1 \cap (U_2 \setminus \{x\}) = \emptyset \end{array} \quad \begin{array}{l} \Gamma \triangleright e_2 : [U_2, m, [\Delta, x : T_1] \blacktriangleright T_2, k] \\ T_1 = \text{list}(-) \end{array}}{\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : [U_1 \cup (U_2 \setminus \{x\}), n, [\Delta] \blacktriangleright T_2, k]} \quad (\text{Let})$$

$$\frac{\Delta(x) = \text{list}(k) \quad l = n + k \quad \Gamma \triangleright e : [U, l, [\Delta, t : \text{list}(k)] \blacktriangleright T, m] \quad x \notin U \setminus \{t\}}{\Gamma \triangleright \text{let } t = x.\text{TL} \text{ in } e : [(U \setminus \{t\}) \cup \{x\}, n, [\Delta] \blacktriangleright T, m]} \quad (\text{LetTL})$$

- 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