

# A program logic for resources and its application to optimisation validation

Alberto Momigliano

Laboratory for Foundations of Computer Science

University of Edinburgh

In collaboration with:

Martin Hofmann, David Aspinall, Lennart Beringer and Hans-Wolfgang Loidl

Dipartimento di Matematica e Informatica, Università' di Udine

29 Novembre 2005

## PCC infrastructure for resource-related properties

---

- How the past shines his light on the future: Mobius (“Mobility, Ubiquity, Security”) is the follow up of MRG (“Mobile Resources Guarantee”), a joint University of Edinburgh / LMU Munich project (2002-2005 Global Computing.)
- The aim was and still 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
- Approach (certifying compilation): translation from user language into machine language derives independently verifiable certificates
- We complement security aspects of PCC (memory safety,...)

## Outline of the talk

---

- Architecture of MRG
- Syntax and semantics of Grail
- Grail's Program Logic
- Certificate Generation
- Validation of compiler optimisations
- Conclusions

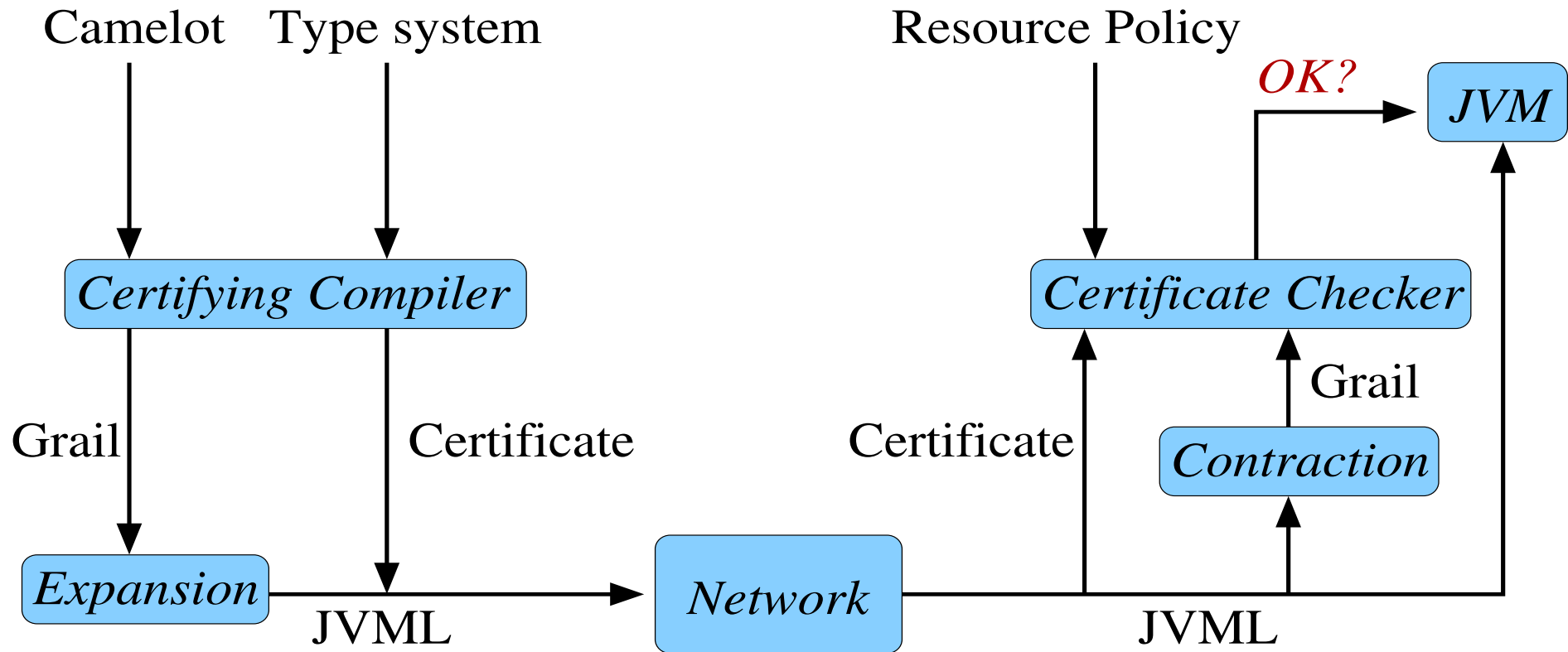
## Components of MRG versus Mobius

---

- We write programs in a custom high-level language **Camelot**, a functional language with an OCaml-like syntax. In Mobius this will be full-fledged Java (no laughing, please).
- Camelot is compiled into **Grail**, a functional intermediate code, which is isomorphic to a subset of JVMIL. Mobius will use a hierarchy of intermediate languages, the bottom one being raw, possibly obfuscated (again, no laughing) bytecode.
- We use an instrumented operational semantics for Grail, reflecting the expansion into JVMIL and based on the abstract notion of **resource algebra**
- One instance is a **cost model** for the JVM: instructions, stack and heap sizes.
- The **Grail Logic** is a program logic expressing (resource) assertions about the operational semantics. In Mobius we are already using the same “VDM” style directly on bytecode.
- Camelot has a **resource type inference system**, which is used to produce proofs in a **logic of derived assertions**. Mobius will similarly use advanced type systems for Java.
- The annotated semantics, logics, and meta-theorems have all been formalised in **Isabelle**, and Isabelle proof scripts are used as our proof transmission format. Mobius is coordinated by INRIA, so no money in guessing which system will be the king of the jungle. Our approach (“script-let”) should be adopted, though.

# MRG architecture

---



- Works because of reversible expansion of Grail into JVMML subset
- The whole picture is parametric in the resource policy agreed between the parts.

# 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)
```

- User's annotation @\_ indicates destructive pattern match: node associated to the Cons is automatically reclaimed into a freelist w/o specific malloc, dispose.
- Linear (or more liberal discipline such as *Usage Aspects* [Aspinall00]) typing prevents from deallocating live variables.
- Whole program compilation where each Camelot function yields one JVM method, including an explicit memory manager independent from JVM's GC – that's the whole point of the exercise.

⇒ Wish to certify memory consumption of compiled output.

## Program analysis, certification & proof checking

---

```
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)
```

- Memory consumption inferred from program annotations using the Hofmann-Jost type system [POPL'02]: and the results are...
- `ins` consumes one memory cell, independent from actual input.
- `sort` does not consume any memory (in-place).
- 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 in Isabelle (or better, in a dedicated proof-checker) before executing program.

## 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: operational model (processor) formalised in higher-order logic that is built on top of Twelf, use Twelf proof terms as certificates

### MRG and hopefully Mobius:

- Formalise *instrumented* operational semantics of (virtual) machine language
- Use a general-purpose program logic (sound, complete & expressive, little automation)
- Derive special logics in proof assistants and see interpreted type inference as certificate generation (i.e. the all elusive specifications).

## Grail: Characteristics

---

- Combine OO-aspects of bytecode (fields, methods) with low-level functional language
- Functional view: first-order functions; no nesting; all free variables in parameters; applications only to values.
- Imperative view: easily convertible into various virtual machines formats
- Coincidence between functional and imperative views makes conversion reversible
- Method-level structure as in JVM

	Functional view	Imperative view
Appel. . .	tail-recursive (1st-order) function	labelled basic block
Lambda-JVM	let-binding, ANF	assignment
Grail	variable	register
	optimisation of variable usage	register allocation
	free variables	liveness
	function call	jump

- Emitted bytecode is highly structured (Leroy's conditions)

## (Subset of the) Syntax of Grail

---

- A Grail program is a list of *methods* each containing a list of tail-recursive *functions*.

$$\begin{aligned} e \in \text{expr} \quad ::= & \quad a \mid \text{prim } p \ a \ a \\ & \quad \mid \text{new } c \\ & \quad \mid x.t \mid x.t := a \\ & \quad \mid \text{let } x = e \text{ in } e \mid e ; e \\ & \quad \mid \text{if } e \text{ then } e \text{ else } e \\ & \quad \mid \text{call } f \mid c.m(\bar{a}) \end{aligned}$$
$$a \in \text{args} \quad ::= \quad x \mid v$$
$$v \in \text{args} \quad ::= \quad \text{null} \mid i$$

- Whole development formalized in Isabelle/HOL:
- program encoded using global tables (functions and methods),
- named syntax and op. semantics based on (finite) maps:

$$\text{env} = \text{name} \Rightarrow \text{val}$$
$$\text{heap} = \text{locn} \mid \rightarrow f \ \text{cname}$$

.....

## Grail: resource-instrumented operational semantics

---

- Based on (impure) big-step functional view:

$$E \vdash h, e \Downarrow (h', v, r)$$

where  $r$  is a *resource value* in some structure...

- A *resource algebra*  $\mathcal{R}$  is a partially ordered monoid  $(\mathbb{R}, 0, +, \leq)$ , where  $0$  is the minimum element, and  $+$  is order preserving on both sides.
- $\mathcal{R}$  has constants in  $\mathbb{R}$  for each expression former:
  - $\mathcal{R}^{\text{int}}, \mathcal{R}^{\text{null}}, \mathcal{R}^{\text{var}}, \mathcal{R}^{\text{prim}}, \mathcal{R}_C^{\text{new}}, \mathcal{R}^{\text{getf}}, \mathcal{R}^{\text{putf}}, \mathcal{R}^{\text{comp}}, \mathcal{R}^{\text{let}}, \mathcal{R}^{\text{if}}, \mathcal{R}^{\text{call}}$
  - an operator  $\mathcal{R}_{C,m,\bar{v}}^{\text{meth}} : \mathbb{R} \rightarrow \mathbb{R}$ .
- Each constant denotes the cost associated to an instruction, which are then composed via the monoidal operation.
- A **dynamic** Res Alg. may depend on input environments and/or heaps to extract or examine values. A **static** one may use the typing context.
- In any case, the resources are a purely “non-invasive”; evaluation of an expression is not affected by the resources consumed in subexpressions.

## R.A. examples: 1/2

	Time	Heap	Frames	MethCnts	MethFreq <sup>ld</sup>	MethGuard
$ \mathcal{R} $	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{MS}(Id)$	$\mathcal{NN}$	$\{tt, ff\}$
$\mathcal{R}^{\text{int}}$	1	0	0	$\emptyset$	(1, 0)	tt
$\mathcal{R}^{\text{null}}$	1	0	0	$\emptyset$	(1, 0)	tt
$\mathcal{R}^{\text{var}}$	1	0	0	$\emptyset$	(1, 0)	tt
$\mathcal{R}^{\text{prim}}$	1	0	0	$\emptyset$	(1, 0)	tt
$\mathcal{R}_C^{\text{new}}$	3	$size(C)$	0	$\emptyset$	(3, 0)	tt
$\mathcal{R}^{\text{getf}}$	2	0	0	$\emptyset$	(2, 0)	tt
$\mathcal{R}^{\text{putf}}$	3	0	0	$\emptyset$	(3, 0)	tt
$\mathcal{R}^{\text{comp}}$	0	0	0	$\emptyset$	(0, 0)	tt
$\mathcal{R}^{\text{let}}$	1	0	0	$\emptyset$	(1, 0)	tt
$\mathcal{R}^{\text{if}}$	0	0	0	$\emptyset$	(0, 0)	tt
$\mathcal{R}^{\text{call}}$	1	0	0	$\emptyset$	(1, 0)	tt
$\mathcal{R}_{C,m,\bar{v}}^{\text{meth}}(r)$	$ \bar{v}  + 2 + r$	$r$	$r + 1$	$r \cup_+ \{C.m\}$	$\text{Freq}_{C.m, \bar{v} }(r)$	$G_{C,m}(\bar{v}) \wedge r$
$0_{\mathcal{R}}$	0	0	0	$\emptyset$	(0, 0)	tt
$+_{\mathcal{R}}$	+	+	$max$	$\cup_+$	$\dagger_{\text{Freq}}$	$\wedge$
$\leq_{\mathcal{R}}$	$\leq$	$\leq$	$\leq$	$\subseteq_+$	$\leq_{\text{Freq}}$	$\leq_{\text{Guard}}$

## R.A. examples: 2/2

---

- The **Time** algebra models an instruction counter that approximates execution time; each expression is charged according to the number of JVM instructions to which it expands
- The **Heap** algebra counts the size of heap space consumed during execution (ignoring the possibility of garbage collection). Only the *new* instruction consumes heap.
- The **Frames** algebra counts the maximal number of frames on the stack during execution.
- The **MethCnts** algebra traces invocations by accumulating a multiset of invoked method names.
- The **MethFreq**<sup>*ld*</sup> algebra calculates a measure of the frequency of calls to the method *ld* (a long identifier  $C.m$ ), by accumulating the maximal period between successive calls;
- The **MethGuard** algebra does not calculate a quantitative resource, but maintains a boolean monitor which checks that arbitrary guards  $G_{C,m}(\bar{v})$  are satisfied at invocations of method  $m$  in class  $C$ .
- Others: more general traces of method invocation, read-write effects on heap locations, live variables, complete traces of heaps during execution, ...
- Traditional static ones: code size, binding-time analysis, strictness. ...

## Operational semantics: some rules

---

$$\frac{E \vdash h, x \Downarrow l, h, r_x}{E \vdash h, x.f \Downarrow h, h(l).f, r_x + \mathcal{R}^{\text{getf}}} \quad (\text{GETF})$$

JVM resources as product of **Time**, **Heap** and **Frame**. In the `getField` case  $\langle 2, 0, 0 \rangle$ .

$$\frac{E \vdash h, e_1 \Downarrow h_1, v_1, r_1 \quad E \langle x := v_1 \rangle \vdash h_1, e_2 \Downarrow h_2, v, r_2}{E \vdash h, \text{let } x = e_1 \text{ in } e_2 \Downarrow h_2, v, r_1 + \mathcal{R}^{\text{let}} + r_2}$$

frame resources wrt LET:  $1 + \max(r_1, r_2)$

$$\frac{\text{eval}_E(a_i) = v_i \quad \{x_i := v_i\} \vdash h, \text{body}_{C,m} \Downarrow h', v, r}{E \vdash h, C.m(\bar{a}) \Downarrow h', v, (\sum_i \text{cost}(a_i)) + \mathcal{R}_{C,m,\bar{v}}^{\text{meth}}(r)}$$

JVM resources  $\mathcal{R}_{C,m,\bar{v}}^{\text{meth}}(\langle t, h, f \rangle) = \langle |\bar{v}| + 2 + t, h, f + 1 \rangle$

## Program logic I

---

- Recent reappraisal of program (Hoare) logics: embeddings in proof assistants (Kleymann, Nipkow), Separation logics (Reynolds, O'Hearn), Java verification (Jacobs, de Boer, von Oheimb), here at QMUL. . .
- For those with a formal implementation: Mostly encoding a' la Kleymann: deep embedding of language, shallow embedding of assertions, with verified soundness and completeness.
- Pragmatic issue: meta-theoretic foundations vs. program verification (automation). In MRG-PCC both issues are important!
- Our judgement take the form:  $G \triangleright e : P$ 
  - $e$  is a Grail expression;
  - $G$  is a context used for storing assumptions for recursive methods and functions;
  - $P$  is an assertion, i.e. a predicate in the meta-logic, merging pre and post-condition.
  - Assertions are simply predicates over semantic values:

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

relating the environment and the initial heaps to the result, the final heap and a the resource value.

## Program logic II: proof rules

---

- No auxiliary variables (usage of pre-heap inspired by hooked variables in VDM)
- Judgements interpreted as partial “correctness” statements: validity  $\models e : P$  defined as  $(E \vdash h, e \Downarrow (h', v, r) \longrightarrow P[E, h, h', v, r])$ . Sample rules:

$$\frac{}{G \triangleright x.f : \{h = h' \wedge (\exists l. E(x) = l \wedge v = h(l).f) \wedge r = \mathcal{R}^{\text{var}} + \mathcal{R}^{\text{getf}}\}}$$

$$G \triangleright e_1 : P_1 \quad G \triangleright e_2 : P_2$$

$$\frac{}{G \triangleright \text{let } x = e_1 \text{ in } e_2 : \{\exists h_1 v_1, r_1 r_2. P_1[E, h, h_1, v_1, r_1] \wedge P_2[E[x := v_1], h_1, h', v, r_2] \wedge r = r_1 + \mathcal{R}^{\text{let}} + r_2\}}$$

$$\frac{G \cup \{(C.m(\bar{a}), P)\} \triangleright \text{body}_{C,m} : P[\{x_i := \text{eval}_E(a_i)\}, h, h', v, \mathcal{R}_{C,m,\text{eval}_E(\bar{a})}^{\text{meth}}(r)]}{G \triangleright C.m(\bar{a}) : P[E, h, h', v, r]}$$

- Structural and admissible rules: context lookup, rule of consequence, CUT.
- $\rightsquigarrow$  Single proof system suffices for mutual recursion and parameter adaptation
- Soundness and completeness hold – For details, see [TPHOLs04].
- Termination Logic is separated and defined on top of the Partial Logic.

## Program logic III: example specification (insertion sort)

---

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

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

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

- $h, r \models_X n$  defined inductively, introduces case-splits during verification
  - Proof rules contain existentials over intermediate heaps and instrumentations
- $\rightsquigarrow$  automatic proof search impractical even after VCGen: automation by compiler difficult
- Certificate Generation: exploit program structure and compiler analysis by proving properties that are more closely related to the type system

## 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;
  - provided that results of the compile-time analysis are communicated as method-level specifications (invariants)

$$\text{ins} : 1, \mathbf{l} \times \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0$$
$$\text{sort} : 0, \mathbf{L}(0) \rightarrow \mathbf{L}(0), 0$$

- Fixed assertion format  $\llbracket \mathbf{U}, n, [\Delta] \blacktriangleright \mathbf{T}, m \rrbracket$  in the (core) logic

$$G_{\text{insort}} \triangleright \text{List.ins} : \llbracket \{\mathbf{a}, \mathbf{l}\}, 1, [\mathbf{a} : \mathbf{l}, \mathbf{l} : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0 \rrbracket$$
$$G_{\text{insort}} \triangleright \text{List.sort} : \llbracket \{\mathbf{l}\}, 0, [\mathbf{l} : \mathbf{L}(0)] \blacktriangleright \mathbf{L}(0), 0 \rrbracket$$

## Proof rules

---

- LFD rule in Camelot (L<sub>Let</sub>):

$$\frac{\Gamma_1, n \vdash M_1 : A, k \quad \Gamma_2, x : A, k \vdash M_2 : B, m}{\Gamma_1, \Gamma_2, n \vdash \text{let } x = M_1 \text{ in } M_2 : B, m}$$

- Note linearity condition for eliminating deallocation of live cells.
- Grail Certificate logic : linear context implemented in two components ( $\Gamma, \mathcal{U}$ ).
- Grail's Proof rule (L<sub>Let</sub>)

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

provided  $\mathcal{U}_1 \cap (\mathcal{U}_2 \setminus \{x\}) = \emptyset$

- *Atomic* rules for [non]destructive match-statements and for invocations of **make**, i.e. the compilation of a match yields a “macro” in the logic.
- 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 (type-like) assertions form.
- Representation: Isabelle/HOL script that invokes a standard tactic **proveMe**.

Consumer side:

- 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, ...)
  - On-going generalization to algebraic data-type.

## An (amusing) application of Res. Alg: optimisation validation

---

- Starting point: proving compiler correctness is hopeless, because of complexity and need to avoid freezing it once verified
- **Translation validation**, [Pnueli98] establishes that a particular instance of a transformation undertaken by an optimising compiler is semantics preserving.
- Validation mechanism that, after every run of a compiler, formally confirms that the target code produced on that run is a correct translation of the source producing:  
    “[. . .] the same result while (*hopefully*) executing in less time or space or consuming less power.” [Rinard00]
- Let's take this seriously and use the ordering provided by the algebra to show that there is indeed an improvement!
- **Optimisation validation**: establish when an optimised program consumes fewer resources than its source.
- Resource usage improvement may even be a more important concern than correctness, because it encompasses the security requirements of the domain (cf. memory safety in PCC).

## Methodology

---

- To consider  $e_1 \longrightarrow e_2$ , an optimisation, we want to establish that the transformation is improving with respect to a cost model, expressed as a resource algebra.
- $E, h \vdash e_1 \Downarrow r_1 \wedge E, h \vdash e_2 \Downarrow r_2 \implies r_2 \leq r_1$  where  $\leq$  refers to the ordering from  $\mathcal{R}$  (cf. Sands' *Improvement Theory*).
- Optimisation sequences:  $P_1 \longrightarrow P_2 \longrightarrow \dots \longrightarrow P_n$ , where each  $P_i \longrightarrow P_{i+1}$  is an optimisation for some algebra  $\mathcal{R}_i$ . Additionally, each step in the optimisation should be non-increasing for the target cost model  $\mathcal{R}$ .
- To state and prove dynamic cost optimisations, we use a program logic to make assertions about functions that bound the resource consumed.

$$ST_1 \triangleright e_1 : \{F_1(E, h) \leq r\} \qquad ST_2 \triangleright e_2 : \{r \leq F_2(E, h)\}$$

- The assertions state that the resource consumed when executing  $P_1$  is bounded from below by some function  $F_1$  of the input heap and env, and that the resources consumed by  $P_2$  are bounded from above by a function  $F_2$ . To show that  $P_2$  is an optimisation of  $P_1$  we must now prove that  $\forall h E. F_2(E, h) \leq F_1(E, h)$

## Some low-level optimisations

---

```
method static int calc0() = let i = 0 in let x = 1 in let y = 2 in let g = 0 in call f
  fun f(int i, int x, int y, int g) = if i < 24 then call h else var g
  fun h(int i, int x, int y, int g) = let j = i + x in let i = j + y in let g = 2 * i in call f
```

```
method static int calc1() = let i = 0 in let x = 1 in let y = 2 in let g = 0 in call f
  fun f(int i, int x, int y, int g) = if i < 24 then call h else var g
  fun h(int i, int x, int y, int g) = let i = i + 3 in let g = 2 * i in call f
```

```
method static int calc2() = let i = 0 in let g = 0 in call f
  fun f(int i, int g) = if i < 24 then call h else var g
  fun h(int i, int g) = let i = i + 3 in let g = 2 * i in call f
```

```
method static int calc3() = let i = 0 in let g = 0 in call h
  fun h(int i, int g) = let i = i + 3 in let g = 2 * i in if i < 24 then call h else var g
```

i	t <sub>i</sub>	Transformation
0	213	
1	197	Constant propagation and constant folding
2	193	Dead assignment elimination
3	176	Branch movement, inlining, redundant test elimination

## Discussion

---

### Future work:

- Generalise existing system of derived assertions (sharing, usage-aspects, separation), and evaluate on bigger examples
- Extract stand-alone proof checker
- Derive certificate generation for other resources: frame stack, time, limits and separation conditions on method parameters
- Investigate under which conditions on transformations static optimisations imply dynamic ones and therefore can be checked by type inference.

### Conclusion:

- MRG-motto: certificate generation by interpreting type-systems in program logic
- 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: