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

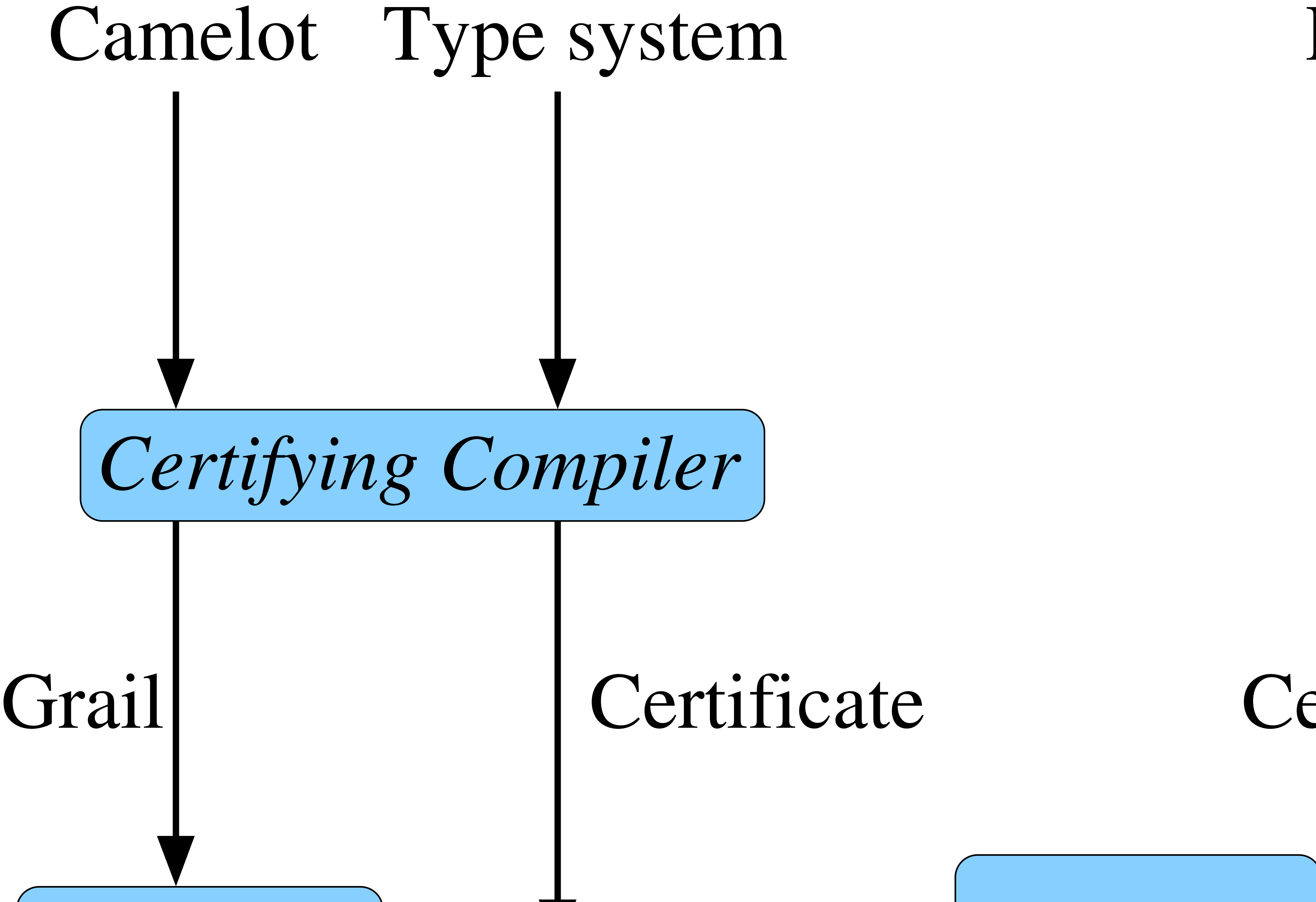
Reasoning Seminar, November, 5th, 2004

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



Camelot

Type system

Certifying Compiler

Grail

Certificate

Ce

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:

```
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: `ins` consumes one memory cell, independent from actual input), `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):

$$\mathit{ST}\ \mathtt{LST.rev}\, z \ = \ \lambda\, E\, h\, h'\, v\, p.\, \forall\, n\, a\, X\, m\, b\, Y.$$

$$\left(\begin{array}{l} (E\ z = [\text{Ref}\ a, \text{Ref}\ b] \ \wedge\ h, a \models_X n \ \wedge\ h, b \models_Y m \ \wedge\ X \cap Y = \emptyset) \\ \longrightarrow |\mathit{dom}(h)| = |\mathit{dom}(h')| \ \wedge\ p = \langle (29n + 13)\ 0\ (n+1)\ (n+1)\rangle \end{array}\right)$$

Verification doable but cumbersome (∃-instantiations, case splits,. . .)

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 $\longrightarrow$ general program logic $\longrightarrow$ 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

  $\leadsto$ 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-pupose 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)

```
datatype expr =
  Int     int
| Primop  (int => int) name name
| New     cname (fldname  name) list
| GetF    name fldname
| PutF    name fldname name
| InvokeStatic cname mname ARGTYPE
| Let     name expr expr
| Ifg     name expr expr
| Call    funame
```

- Program encoded using global tables (functions and methods)
- Impure functional semantics based on (finite) maps:

```
env   = name => val
heap = locn |->f cname
       fldname => locn => val
       cname => fldname => ref
```

Grail: resource-instrumented operational semantics

Based on (impure) functional view:

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

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

$$\frac{E \vdash h, e_1 \Downarrow (h_1, w, p) \quad w \neq \bot \quad E\langle x := w\rangle \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \Downarrow (h_2, v, \mathcal{P}^{\texttt{let}}(x, p, q))} \quad (\text{LET})$$

$$\frac{E\langle x\rangle = \mathit{Ref}\ l}{E \vdash h, x.t \Downarrow (h, h(l).t, \mathcal{P}^{\texttt{getf}}(x, t))} \quad (\text{GETF})$$

where $\mathcal{P}^{\texttt{getf}}(x, t) = \langle 2\ 0\ 0\ 0\rangle$ 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 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 $\models e : A$ defined as

$$\forall\, E\, h\, h'\, v\, p.\quad (E \vdash h, e \Downarrow (h', v, p) \longrightarrow A\, E\, h\, h'\, v\, p).$$

- Termination considered orthogonal

Program logic III: proof rules

$$\frac{\Gamma \rhd e_1 : A_1 \quad \Gamma \rhd e_2 : A_2}{\Gamma \rhd \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \lambda\, E\, h\, h'\, v\, p.\, \exists\, p_1\, p_2\, h_1\, w.\ \begin{array}{l} (A_1\, E\, h\, h_1\, w\, p_1) \;\wedge\; w \neq \bot \;\wedge \\ (A_2\, (E\langle x := w\rangle)\, h_1\, h'\, v\, p_2) \;\wedge \\ p = \mathcal{P}^{\texttt{let}}(x, p, q) \end{array}}$$

(VLET)

$$\frac{}{\Gamma \rhd x.t : \lambda\, E\, h\, h'\, v\, p.\, \exists\, l.\ \begin{array}{l} E\langle x\rangle = \mathit{Ref}\ l \;\wedge\; h' = h \;\wedge \\ v = h'(l).t \;\wedge\; p = \mathcal{P}^{\texttt{getf}}(x, t) \end{array}}$$

(VGETF)

- Structural rules: context lookup and rule of consequence
- Admissable rules (derived in Isabelle): cut
- Context Γ stores recursive assumptions. ⇝ 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. $\rightsquigarrow$ incompleteness of meta-logic (HOL) is inherited by program logic
- Completeness proven by defining strongest specifications, a specification table $\widehat{ST}$ associating to each function call / method invocation its strongest specification, proving that the corresponding context is *good* w.r.t. $\widehat{ST}$, and applying (a variant of) the cut rule and MUTREC.

Program logic V: example specification (insertion sort)

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

Lemma: *insSpec* $\wedge$ *sortSpec* $\longrightarrow\ \rhd$ List $\diamond\ \mathit{sort}([xs])$: *MS* List $\mathit{sort}\ [xs]$

- $h, r \models_X n$ 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
- $\leadsto$ 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

Insertion sort: compiler output

```
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** $\diamond$ **make**$(\ldots)$: takes object from freelist, or calls **new**
- **D** $\diamond$ **free**(x): inserts object into freelist
- **D** $\diamond$ **main**(l): constructs initial freelist, calls `List` $\diamond$ `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` $\diamond$ `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 **start**`: string list -> unit`

  Output: a *linear function* s such that **start**$(\mathtt{l})$ will not call **new** when evaluated in a heap h where

  - $\mathtt{l}$ 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 $\mathtt{l}$
  - the freelist has length not less than $s(n)$
- How does this work?
  - Annotate types with freelist annotations for each constructor: **iTree**(n, m)
  - Judgements $\Gamma, n \vdash e : T, m$ 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 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 freelistrepresentation 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

  $\rightsquigarrow$ proof rules are expressed at a level where program variables occur (affinely) linear

- Linear context implemented in two components
- Example rule (Let)

$$\frac{G \rhd e_1 : [\![U_1, n, [\Gamma] \blacktriangleright S, k]\!] \qquad G \rhd e_2 : [\![U_2, k, [\Gamma, x : S] \blacktriangleright T, m]\!]}{G \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : [\![U_1 \cup (U_2 \setminus \{x\}), n, [\Gamma] \blacktriangleright T, m]\!]}\ U_1 \cap (U_2 \setminus \{x\}) = \emptyset$$

- Atomic rules for (destructive and non-destructive) match-statements and for invocations of **make**
- Example rule ($\mathit{ListMatchD}$)

$$\frac{\Gamma(x) = \mathbf{L}(k) \qquad G \rhd e : [\![U, n + k + 1, [\Gamma, h : \mathbf{I}, t : \mathbf{L}(k)] \blacktriangleright T, m]\!] \qquad x \notin U \cup \{h, t\}}{G \rhd \texttt{let}\ h = x.\mathsf{HD}\ \texttt{in}\ \texttt{let}\ t = x.\mathsf{TL}\ \texttt{in}\ \mathbf{D} \diamond \mathbf{free}(x)\,;\, e : [\![(U \setminus \{h, t\}) \cup \{x\}, n, [\Gamma] \blacktriangleright T, m]\!]}$$

- 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