An overview of the Mobile Resource Guarantees Project

Lennart Beringer
Laboratory for Foundations of Computer Science
University of Edinburgh

Funded by the EC, under initiative “Global Computing”,IST-2001-33149

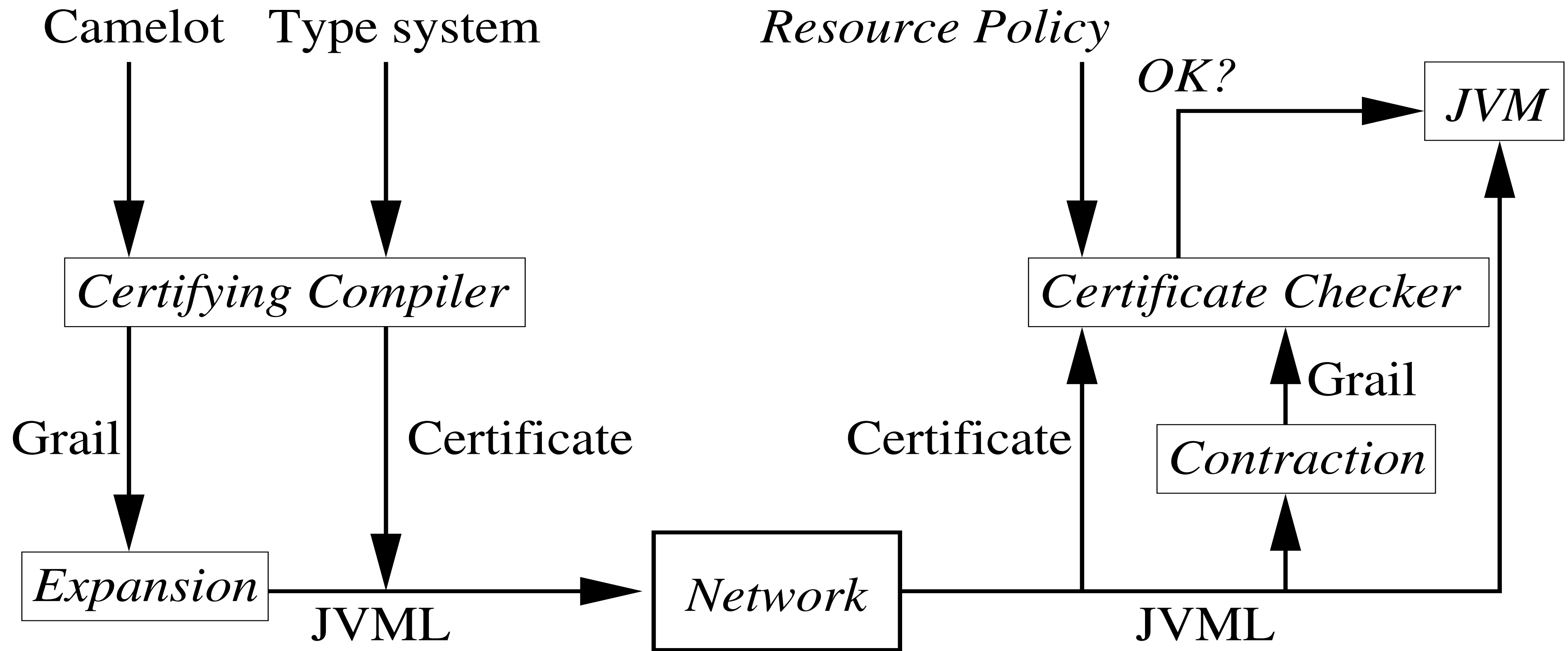
TYPES 2004

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
- PCC: code consumer requires transmitted (low-level) program to come with automatically verifiable proof that his resource policy is fulfilled
- Certifying compilation: translation from high-level language into machine language (semi-automatically) derives independently verifiable certificates
- MRG: certificate generation by type systems for resource usage and consumption
- Complements applications of PCC that target memory (type) safety

This talk: brief overview, short demo, conclusion

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: (monomorphic) 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 (initial length of freelist).

Heap analysis & certification

```
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 using Hofmann-Jost's 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
- Certificate: encoding of the result of the type inference as method specifications

Insertion sort: compiler output

method static public $\texttt{List ins}(\textbf{int}\ \texttt{a},\ \textbf{D}\ \texttt{l}) =$ ...**D.make**$(\texttt{a}, \textbf{null})$...

method static public $\texttt{List sort}(\textbf{D}\ \texttt{l}) =$

if $l =$ **null then** $\texttt{null}$

else $\texttt{let h} = \texttt{l}.\text{HD}\ \texttt{in let t} = \texttt{l}.\text{TL}\ \texttt{in let _} = \textbf{D.free}(\texttt{l})\ \texttt{in}$

$\texttt{let l} = \texttt{List.sort}(t)\ \texttt{in List.ins}(\texttt{h}, \texttt{l})$

...plus code for memory management and runtime environment methods

- **D.make**$(\ldots)$: takes object from freelist, or calls **new**
- **D.free**(x): inserts object into freelist
- **D.main**(l): constructs initial freelist, calls $\texttt{List.sort}(\textbf{s2i}(l))$

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 $\texttt{List.sort}(l)$, 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 **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, does not overlap with $\mathtt{l}$, and has length not less than $s(n)$

- How does this work?

  - Annotate types with freelist annotations for each constructor: $\mathbf{L}(k)$
  - 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 some method for preventing deallocation of live cells (linear typing, usage aspects, layered sharing,. . .)

- Example: List.ins : $1, \mathbf{I} \times \mathbf{L}(0) \to \mathbf{L}(0), 0$ List.sort : $0, \mathbf{L}(0) \to \mathbf{L}(0), 0$

Consumer-side verification

- Formalise operational semantics of (virtual) machine language (like FPCC)
- Use a general-purpose program logic: soundness & (relative) completeness formally proven, little automation
- Derive special logics formally in theorem prover:
  - Interprete high-level typing judgements as a formulae in the core logic (fixed assertion format), guided by correctness statement of type system
  - Derive lemmas for low-level code fragments in the core logic. Guiding motivation: $\Gamma, n \vdash e : T, m \rightsquigarrow \rhd \textit{compile}(e) : [\![\Gamma, n, T, m]\!]$

    But: in principle, we don't need to prove that the implication holds
  - Implement tactic that "replays" typing derivations
  - Syntax-directedness of rules limits necessity for proof search
  - Simple side conditions: (in-)equalities over numbers, set inclusions...
- Invariants (method specifications) supplied by producer via certificate
- Expansion of statements in fixed assertion format into core logic (and op. semantics) always possible, but usually not necessary (TCB negotiation)

Demo

- JVM integration
  - Modified security manager: execution permitted only after successful verification
  - Hand-crafted Isabelle-files for program representation and certificate
- Prototypical implementation of certificate generation
  - Not yet fully finished (needs debugging, evaluation, polishing . . . !!!)
  - Examples: standard functions over (integer) lists: insertion sort, append, reverse, double. More functions (heapsort, flatten. . .) have been verified for manually created certificates
  - producer: input: prog.cmlt output: prog.class & Certificate.thy (contains method specifications and flow information)
  - consumer: input: prog.class output: Consumer1.thy (contains tactic and verification of method bodies) & Consumer2.thy (contains top-level correctness theorems, with standard proofs)

Discussion

Current & future work:

- Full integration of software components (CertGen, MRG-JVM, ...)
- Resource policies (wrapper, other policies)
- Generalise existing system of derived assertions (sharing, usage-aspects, separation), and evaluate on bigger examples
- Extract stand-alone proof checker from Isabelle tactic and derived proof rules
- Derive specialised logics and certificate generation for other resources: frame stack, time, limits and separation conditions on method parameters
- Termination logic & certification

Conclusion:

- MRG-motto: use structure available in HLL: program structure & type structure
- CertGen by interpreting (resource) type systems in program logic
- Chain of abstractions: operational semantics $\longrightarrow$ general program logic $\longrightarrow$ derived specialised logics with automation
- Development backed up by implementation in Isabelle/HOL

Extra slide: Certificates and automated verification

Producer-generated certificate:

- Content: method-level specifications in derived-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, . . .)