Mobile Resource Guarantees: Resource Bounds for Functional Languages

Hans-Wolfgang Loidl

Ludwig-Maximilians Universität, München

29th October 2003

http://www.lfcs.ed.ac.uk/mrg

Overview

Structure of the talk:

- Overview of the MRG project
- High-level programming language: Camelot
- Inference of heap consumption
- Program logic
- Embedded Systems Language: Hume
- Conclusion

I. Mobile Resource Guarantees

Objective:

Proof-carrying code for resource-related properties, where proofs are generated from typing derivations in a resource-aware type system.

Partners:

Ludwig-Maximilians Universität, München,

(Prof M. Hofmann, S. Jost, H-W. Loidl, O. Shkaravska)

Edinburgh University

(Prof D. Sannella, D. Aspinall, L. Beringer, S. Gilmore, K. MacKenzie, I. Stark)

This work is funded by the EU under the IST-FET project *Mobile Resource Guarantees* No. IST-2001-33149.

Why is this useful?

Restrict the execution of mobile code to those adhering to a certain resource policy.

Application Scenarios:

- A user of a handheld device might want to know that a downloaded application will definitely run within the limited amount of memory available.
- A provider of computational power in a Grid infrastructure may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption.

Advantages of PCC

Current forms of authentication:

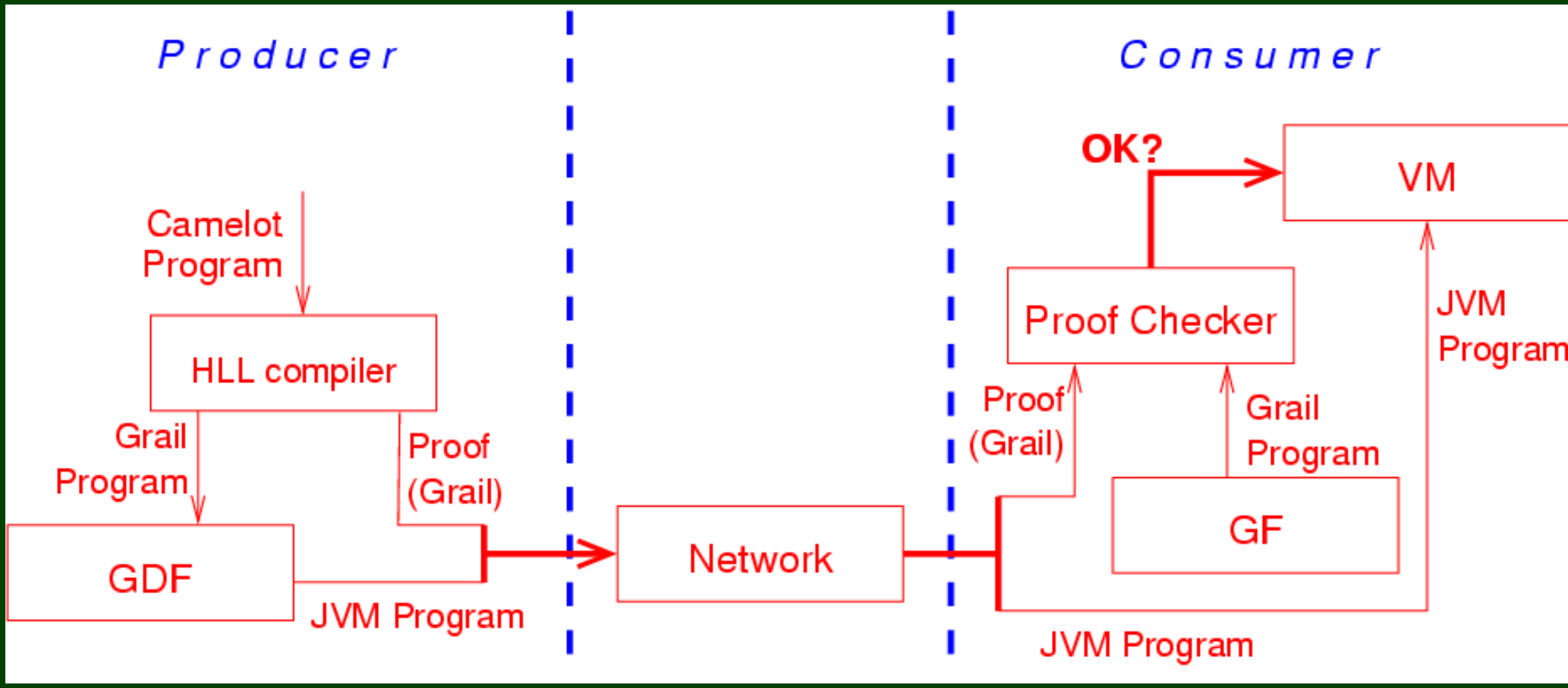
- Java: originally sandbox model: all code is untrusted; since version 1.2: security policies managed through cryptographic signatures.
- Windows: Microsoft Authenticode uses cryptographically signed code.

These methods say nothing about the code itself, only its supplier!

Proof-carrying-code gives guarantees about code behaviour via a condensed formal proof

- Checked by client before execution
- Unforgeable tamper-proof and independent of trust networks
- Proofs may be hard to generate, but are easy to check

A Proof-carrying-code Infrastructure



II. High-level Programming Languages

In MRG we use Camelot (Shkaravska, Hofmann, 2002) an ML-like functional language with the following features

- strict, polymorphic
- first order (but functions may be passed as arguments)
- extensions for explicit control of heap-allocated data
- object-oriented extensions
- compiled to JVM bytecode

Why a functional language?

- extended type systems can guarantee that in-place operations are safe;
- static analyses can infer resource consumption of the program;

Example: insertion sort

Insertion sort over list of integer values.

```
type ilist = Nil | Cons of int*ilist

let insert x l =
    match l with Nil -> Cons(x,Nil)
               | Cons (h,t) ->
                   if x<=h then Cons(x, l)
                           else Cons(h, insert x t)

let sort l =
    match l with Nil -> Nil
               | Cons (h,t) -> insert h (sort t)
```

In-place Operations via a Diamond Type

Using operators, such as `Cons`, amounts to heap allocation.

Additionally, Camelot provides extensions to do in-place operations over arbitrary data structures via a so called diamond type $\diamond$ with $d \in \diamond$:

```
match l with Nil@d => e1
             | Cons (h,t)@d => ... Cons (x,t)@d ...
```

The memory occupied by the cons cell can be re-used via the diamond `d`.

Note:

- $\diamond$ is an abstract data-type
- structured use of diamonds in branches of pattern matches

Camelot's Memory Management Model

The implementation of Camelot uses a special diamond class that can contain members of any data-type in the program (MacKenzie, Wolverson, 2003).

Camelot uses a 2-level heap model:

- the L1-heap is explicitly managed by the Camelot compiler, based on information given by the diamond type;
  the compiler maintains a freelist of diamond objects.
- the L2-heap is the heap managed by the JVM;
  if the freelist is empty a new object is allocated in the L2-heap.

How does this fit with referential transparency?

Using a diamond type, we can introduce side effects:

```
type ilist = Nil | Cons of int*ilist

let insert1 x l =
    match l with Nil -> Cons (x, l)
                 | Cons(h,t)@d ->
                     if x <= h then Cons(x, Cons(h,t)@d)
                                else Cons(h, insert1 x t)@d

let sort l =  match l with Nil -> Nil
                           | Cons(h,t) -> insert1 h (sort t)
```

Now, what's the result of

```
let start args =
    let l = [4,5,6,7] in
    let l1 = insert1 6 l in
    print_list l
```

Linearity saves the day

We can characterise the class of programs for which referential transparency is retained.

Theorem: A linearly typed Camelot program computes the function specified by its purely functional semantics (Hofmann, 2000).

But: linearity is too restrictive in many cases; we also want to use diamonds in programs where only the last access to the data structure is destructive.

More expressive type systems to express such access patterns are readonly types (Aspinall, Hofmann, Konecny, 2001) and types with layered sharing (Konecny 2003).

As with pointers, diamonds can be a powerful gun to shoot yourself in the foot. We need a powerful type system to prevent this, and want a static analysis to predict resource consumption.

III. Inference of Heap Consumption

We have a heap space inference for Camelot programs that produces judgements like this

$$\{x : L(L(B, 1), 2), 3\} \vdash e : L(B, 4), 5$$

meaning "to evaluate the program expression e in a context where variable x is bound to a list $[x_1, \ldots, x_m]$, heap space of size $3 + 2m + 1\Sigma_i \mid x_i \mid$ is required and free heap space of the size $5 + 4 \mid l \mid$ is left over after producing the result l."

The inference is type-system-based, and works over algebraic data structures.

Rule for function calls

A, B, C are types, $k, k', n, n' \in \mathbb{Q}^+$, f is a (program) function and $x_1, \ldots, x_p$ are variables, Σ is a table mapping function names to types.

$$\frac{\Sigma(f) = (A_1, \ldots, A_p, k) \longrightarrow (C, k') \qquad n \geq k \qquad n - k + k' \geq n'}{\Gamma, x_1 : A_1, \ldots, x_p : A_p, n \vdash f(x_1, \ldots, x_p) : C, n'} \quad (\text{FUN})$$

Inference Example: Insertion Sort

Recall the definition of insertion sort:

```
type ilist = Nil | Cons of int*ilist

let insert1 x l =
    match l with Nil -> Cons (x, l)
                 | Cons(h,t)@d ->
                     if x <= h then Cons(x, Cons(h,t)@d)
                                else Cons(h, insert1 x t)@d

let sort l =  match l with Nil -> Nil
                         | Cons(h,t) -> insert1 h (sort t)
```

The inference gives the following types:

```
insert: <1>, int -> ilist(<0>|int,$,<0>) -> ilist(<0>|int,$,<0>), <0>
sort: <1>, ilist(<0>|int,$,<1>) -> ilist(<0>|int,$,<0>), <0>
```

This says that the call `insert x l` requires 1 heap cell plus 0 heap cells for each `Cons` of the input list. At the end it will leave 0 heap cells per `Cons` of the result list.

Characteristics of the Inference

- Type-system based, with annotations for space consumption.
- Information of both the heap required for evaluation and the available heap after evaluation is provided.
- Type annotations give different weight to the constructors.
- Annotated types represent linear functions of heap space over the data structures.

Main Results about the Inference

The inference

- is proven correct for a core language similar to desugared Camelot (Hofmann, Jost 2003);
- generalises to algebraic data structures;
- has been implemented in the Camelot compiler;
- is efficient using linear-program solving;
- scales well over the program size: example with 500 mutually recursive functions, yielding 10117 inequalities over 6608 variables, generated in 0.63 seconds and solved in 45.7 seconds.

IV. Program Logic

Goal:

Prove resource properties on a low-level, JVM-like language for mobile code using a proof-carrying code approach.

In particular:

- Resources: time, space, system calls
- General low-level language modelled after JVM bytecode: Grail
- Proof-carrying code: mobile code is sent together with a proof of correctness

Example: in-place list reversal

Functional reading of the Grail program:

```
method static List rev (List l, List acc) =
let fun f(List l, List acc) =          // Local function declaration
      let val tag = getfield l TAG     // Access to object discriminator
      in if tag = 0 then acc
         else f1(l, acc)               // Conditional function call
      end
    fun f1(List l, List acc) =         // Another local function
      let val h = getfield l HD
          val t = getfield l TL
          val _ = putfield l TAG 1   // RHS with side effect
          val _ = putfield l HD h
          val _ = putfield l TL acc
          val acc = l
          val l = t
      in f(l,acc) end                  // Tail call satisfying parameter conditi
in f(l, acc) end                       // Main expression
```

Grail: Guaranteed Resource Allocation Intermediate Language

- Abstraction over JVM, with possible expansion to other VMs such as .NET
- Translation between JVM bytecode and Grail is reversible
- Two equivalent semantics:
  - imperative: subset of JVM language in a restricted form;
  - functional: big-step call-by-value semantics, with side-effecting operations;
- Restrictions:
  - Methods are represented as set of mutually tail-recursive first-order functions (one function for each basic block)
  - in functions, the name of the formal parameter must be the same as the arguments
  - all intermediate values are named (A-NF)
  - Full λ-lifting
- Grail is the target language for the Camelot compiler

Formalisation of Grail

We define a resource-aware operational semantics for Grail, with judgements of the form

$$\eta \vdash h \xrightarrow{e}_{n} (h', v, \rho)$$

meaning "starting with a heap h and a variable enviroment η, the Grail code e evaluates in n steps to the value v, yielding the heap h' as result and consuming ρ resources."

Resources are modelled as resource tuples of the form

$$\rho \in \mathit{resrec} = \{\, \mathit{clock} : \mathit{nat}, \mathit{callcount} : \mathit{nat}, \mathit{invokedepth} : \mathit{nat}, \mathit{maxstack} : \mathit{nat} \,\}$$

Operational Semantics: Call-rule

$$\frac{\eta \vdash h \xrightarrow{\mathit{body}_{\mathtt{f}}}_{n} (h_1, v, \mathbf{p_1})}{\eta \vdash h \xrightarrow{\mathtt{Call\ f}}_{n+1} (h_1, v, \langle \mathbf{1\ 1\ 0\ 0} \rangle \oplus \mathbf{p_1})} \quad (\textsc{call})$$

Program Logic for Grail

We define a program logic to prove (resource) properties about Grail programs. Judgements in the program logic have the form

$$\mathrm{G} \rhd \mathrm{e} : \mathrm{P}$$

meaning "in the context of the specifications in G, the Grail code e satisfies the specification P."

Specifications are modelled as sets, relating pre-state with post-state:

$$\mathit{vdmassn} = (\mathit{env} \times \mathit{heap} \times \mathit{heap} \times \mathit{val} \times \mathit{resrec})\ \mathit{set}$$

Axiomatic Semantics: Call-rule

$$\frac{(G \cup (\texttt{Call}\ \texttt{f}, P)) \rhd (\textit{funtable f}) : \{(E, h, hh, v, \mathbf{p}).\ (E, h, hh, v, \langle \mathbf{1\ 1\ 0\ 0} \rangle \oplus \mathbf{p}) \in P\}}{G \rhd (\texttt{Call}\ \texttt{f}) : P} \quad (\text{VCALL})$$

Proving Resource Properties

For the in-place list reversal algorithm, no heap space is allocated and the time consumption is linear over the length of the input list `l`.

$$\triangleright\ \texttt{Call rev} : \{\ (E, h, h', v, p) \mid \forall L\, X\, l_1\, A\, Y\, l_2.$$
$$(E\langle \texttt{l} \rangle = \mathit{Ref}\, l_1 \wedge E\langle \texttt{acc} \rangle = \mathit{Ref}\, l_2 \wedge$$
$$h, l_1 \models_X L \wedge h, l_2 \models_Y A \wedge X \cap Y = \emptyset)$$
$$\Longrightarrow (|\mathit{dom}\,(h)| = |\mathit{dom}\,(h')| \wedge p = \langle (31L + 11)\ (L + 1)\ 0\ 0 \rangle)\}$$

Characteristics of the Program Logic

The main characteristics of our resource-aware program logic for Grail are:

- VDM-style judgements of the form $G \triangleright e : P$ rather than Hoare triples of the form $G \triangleright \{P\}\, e\, \{Q\}$.
- The logic formalises resource consumption via resource tuples
- Shallow embedding of the assertion language, i.e. we use the meta-language of the theorem prover to formalise assertions

Main Results for the Program Logic

Results summarised in (Aspinall, Beringer, Hofmann, Loidl, Momigliano 2003):

- Proofs of soundness and completeness for this logic
- Soundness for an extension including dynamic method invocation
- Completely formalised and proven in Isabelle/HOL
- Case studies of space and time consumption for Grail programs

Why can we achieve completeness where other OO-logics cannot?

- We do not have higher-order heap
- This is due to our language being class-based (no dynamic method update)
- We use the full power of the prover in the assertion language.

V. Embedded Systems Language: Hume

Hume (Hammond, Michaelson, 2002) has been designed as a language for resource bounded computation in embedded and distributed systems.

Main features of Hume

- predictable time and space consumption
- separates 3 levels:
  - a declaration layer
  - a coordination layer, and
  - a computation layer;
- execution model: boxes being connected via a coordination language
- computation language is restricted by design to ensure predictability of resources

Example: Parity

```
type bit = word 1;  type parity = boolean;

box even_parity
in  ( b  :: bit,    p :: parity )
out  (p’ :: parity, show :: string)
unfair
  ( 0, true  ) -> ( true, "true" )
| ( 1, true  ) -> ( false, "false" )
| ( 0, false ) -> ( false, "false" )
| ( 1, false ) -> ( true, "false" );
```

Resource Bounded Computation in Embedded Systems

We plan to combine the efforts in the MRG and Hume projects to develop a system for resource bounded computation in embedded systems.

- Uses Hume with 3-level structure as programming language
- Integrates Camelot as a computation language
- Applies inference technology for inferring space (and time) consumption
- Maps the generated code down to machine code
- Application domain: embedded systems, in particular autonomous vehicles

VI. Conclusions

MRG works towards resource-safe global computing.

Key technologies in our infrastructure are

- Proof-carrying-code
- Resource-aware program logic and automated theorem proving
- Extended type systems and type based inference of memory consumption
- Functional programming language, with extensions for in-place operations

Future developments:

- Automatically generate certificates by the Camelot compiler
- Case studies: Camelot cellphone application, running on a cut-down JVM
- Integration with Hume
- Work on proof-preserving program transformations