Implicit Computational Complexity

Simone Martini

Dipartimento di Scienze dell'Informazione
Università di Bologna
Italy

Bertinoro International Spring School
for Graduate Studies in Computer Science,
6–17 March, 2006

Outline: third part

Logic and Programming Languages

Challenges

From Logic to Programming Languages

- How can a host machine assure the amount of resource needed to run a mobile program? A resource-aware type system or program-logic would provide implicit and verifiable certificates.
- In the realm of (first-order) term-rewriting systems, techniques like **quasi interpretations** have been shown to be useful for inferring complexity properties of programs (Bonfante et al.).
- **Type-systems** derived from non-size increasing computations have been exploited in the context of mobile resource guarantees (Hofmann et al., Beringer et al.).
- Enforcing resource-awareness in programming languages is not an easy task. The additional control provided cannot come at the price of unacceptable restrictions to programs.

Inferring Linear Bounds on Heap Size – Hofmann & Jost

- **Language**: first-order functional programming language with recursion; explicit memory management with freelist.
- **Type-system**: simple types, including lists, with resource annotations.
- Goal of the resource annotation is to derive a linear relation between the memory used to represent the input and the memory needed to complete the task.
- Example: Consider a program `P : string list -> unit`
  - We want a linear relation $s(n) = an + b$ with the following meaning:
  - If we evaluate (the compiled) `P` on a input list of lenght n
  - Then, the program will not get stuck from insufficient memory availability
  - Provided that we have a freelist containing initially at least $s(n)$ cells.

Inferring Linear Bounds on Heap Size, II

- The example of the previous slide would get a type
  `P : L(string,a),b -> unit`
- "*If the input list has lenght n, then* `P` *needs $an + b$ cells in the freelist*"
- In general, we need memory assertions also in the result type
- Example: $x : L(B, 2), 3 \vdash e : L(B, 4), 5$
  means
  - if we evaluate e starting with x bound to a list $[u_1, \ldots, u_m]$,
  - and we have a free-list of at least $2m + 3$ cells,
  - then the computation will not get stuck from insufficient memory availability;
  - moreover, if the result is a list $[v_1, \ldots, v_n]$, then at the end the free-list will have at least $4n + 5$ cells.

Inferring Linear Bounds on Heap Size, II

- **Type-system**: Contraction can only be done splitting the corresponding resource annotations: for example, from

  $$x : \mathsf{L(B,3)}, y : \mathsf{L(B,6)} \vdash e : C, 7$$

  we can derive

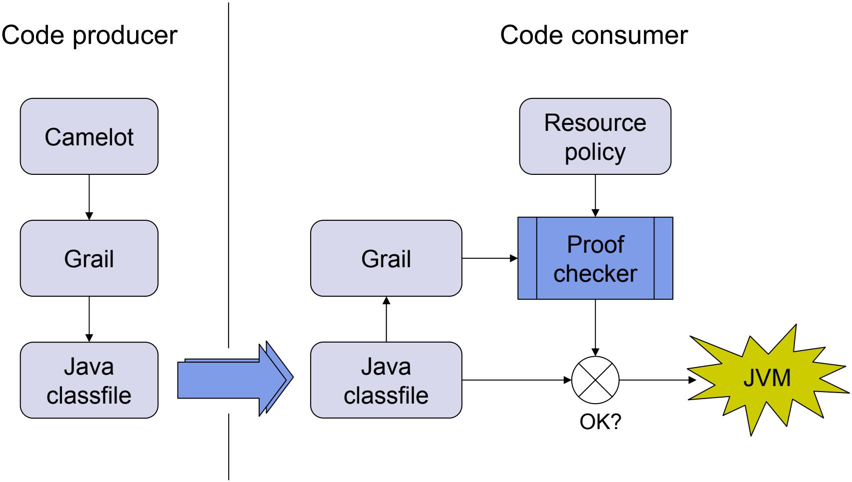
  $$z : \mathsf{L(B,9)} \vdash e\{z/x, z/y\} : C, 7$$

- **Decorations**: given a skeleton of a type derivation (types, but not resource annotations) for e,
  a set of linear inequalities $\mathcal{L}(e)$ is derived.
  Solutions to $\mathcal{L}(e)$ are in one-to-one correspondence with valid type derivations for e.

An example: Mobile Resource Guarantees

- In 2002-2005 a EU funded project tried to embed some of the techniques we discussed in a software architecture.
- MRG, a joint Edinburgh / LMU Munich project funded under the Global Computing pro-active initiative
- Based on the notion of proof carrying code (Necula, 1997):
  - A high-level functional language with a type system ensuring certain bounds on resources
  - A certifying compiler maps programs and their type annotation to a target language, packaging together the code and a (compact version of the) proof that it satifies the required bounds
  - Such packages are unforgeable and tamper evident
  - Clients of the code (e.g., over an untrusted network) receive the package and check the proof before executing the code
  - Checking proof is simple (vs building the proof, which may be hard)

The architecture of MRG



Code producer

Camelot

Grail

Java classfile

Code consumer

Resource policy

Grail

Proof checker

Java classfile

OK?

JVM

Camelot

- Camelot is a high-level functional language, based on OCaml
- Polymorphic types à la ML
- Compiled (through Grail) into standard Java bytecode
- Memory model: freelist, managed directly by the compiled code (as opposed to just rely on garbage collection)
- Programs in Camelot are subjected to space analysis, to express heap usage and linear relations between input/output memory usage

Example

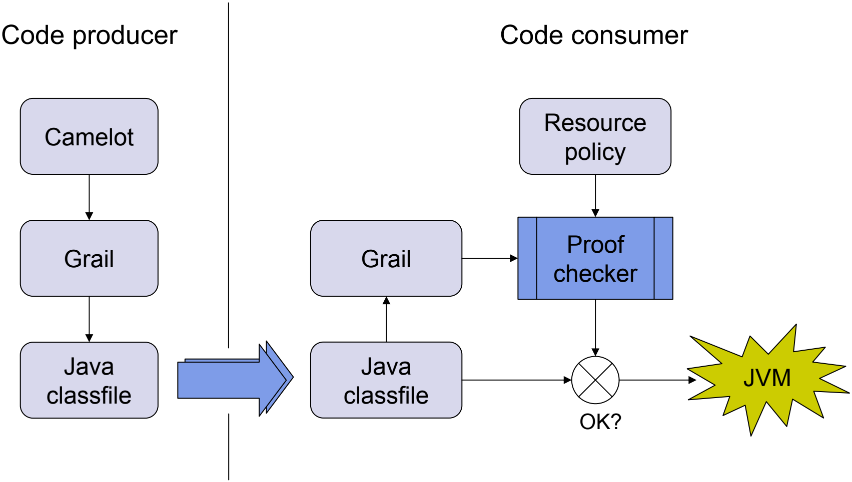
```
type iList = !Nil | Cons of int * iList

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)

let show_list0 l = match l with
                     Nil -> ""
                   | Cons(h,t) -> begin
                       match t with
                         Nil -> string_of_int h
                       | Cons(h0,t0) -> (string_of_int h) ^ ", " ^ (show_list0 t)
                     end

let show_list l = "[" ^ (show_list0 l) ^ "]"

let stringList_to_intList ss =
        match ss with
          [] -> Nil
        | (h::t) -> Cons((int_of_string h),(stringList_to_intList t))

let start args =
        let l1 = (stringList_to_intList args)
     in let _  = print_string ("\nInput list:\n l1 = " ^ (show_list l1))
     in let l2 = sort l1
     in let _  = print_string ("\nResult list:\n l2 = " ^ (show_list l2))
     in ()
```

Fig. 1. A standalone Camelot program

```
ins            : 1,  int -> iList[0|int,#,0] -> iList[0|int,#,0], 0;
int_of_string : 0,  string -> int, 0;
print_string  : 0,  string -> unit, 0;
show_list      : 0,  iList [0|int,#,0] -> string, 0;
show_list0     : 0,  iList [0|int,#,0] -> string, 0;
sort           : 0,  iList [0|int,#,1] -> iList[0|int,#,0], 0;
start          : 0,  list_1 [string,#,2|0] -> unit, 0;
stringList_to_intList : 0,  list_1 [string,#,2|0] -> iList[0|int,#,1], 0;
string_of_int : 0,  int -> string, 0;
```

Fig. 2. Output of space analysis on the program in F

The architecture of MRG



Code producer

Code consumer

Camelot

Grail

Java classfile

Resource policy

Grail

Proof checker

Java classfile

OK?

JVM

Grail

- It is the target of the Camelot compiler, which performs a resource exact compilation

  That is, compilation preserves non only meaning, but also resource behaviour.
- It is the vehicle for proof-carrying code:
  - It is the basis to which to attach the resource assertions
  - It is amenable to formal proofs about resource usage
  - It is the format for sending and receiving guaranteed code
- It can be assembled to (and dissambled from) standard JVM classfiles

Bytecode logic of resources

- The logic allowing to state and prove that the Grail bytecode satify the resource usage
- The construction of proofs uses the type annotations
- Verification is much easier
- But we are not concerned here with this issues...

At the end of this series of lectures...

Many challenges remain. . .

Challenges

- The area of implicit computational complexity appears very fragmented, with many different proposals.
- It is very difficult to compare relative intensional expressive power.
- It is not usually the case a system can be extended with new features preserving its quantitative properties
- Defining just another characterization of polynomial time is not enough.
- Importing these results into the design of (even academic) programming languages is extremely difficult (especially for time bounds).
- Deep, foundational results are extremely needed.


"That's all Folks!"