

# Towards resource control for synchronous cooperative threads

Roberto AMADIO

*Université de Provence (LIF, UMR CNRS 6166)*

## Goal

Static prediction of resources needed for the execution of a program/system.

## An applicative scenario: mobile code in embedded systems

- Programmable switches.
- Application threads in a smart card.

## Preliminary remarks

**Dynamic vs. static** Can always go dynamic... but static is better for

- Efficiency.
- Debugging.

In practice, a combination of the two:

static analyses + code instrumentation

Here the focus is on the static analyses.

## Hard vs. Soft Resource Control

- Functional bounds vs. constant bounds.
- General programs vs. loop programs.

Contrast with *Worst Case Execution Time* technology.

## Thesis for this talk

Resource Control = Termination + Functional bounds on data size

## Plan

1. Cobham's theorem:  $\text{PTIME} = \text{Bounded recursion on notation}$ .
2. Generalisations of Cobham's theorem.
3. Synthesis of bounds on data size.
4. Generalisation to a cooperative synchronous model.
5. Pushing the verification at byte code level.
6. Ongoing work in the **CRISS** project.

## Part 1: Cobham's theorem

A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II*, North Holland, 1965.

## A first-order functional language

- Inductive types

$$t = \dots \mid \mathbf{c} \text{ of } t_1, \dots, t_n \mid \dots$$

- Values, patterns, expressions

$$v ::= \mathbf{c}(v, \dots, v)$$

$$p ::= x \mid \mathbf{c}(p, \dots, p)$$

$$e ::= x \mid \mathbf{c}(e, \dots, e) \mid f(e, \dots, e)$$

- Function definitions by **pattern matching** and **evaluation by value**.

$$\begin{array}{lll} f = & p_{11}, \dots, p_{1,n} & \Rightarrow e_1 \\ & \dots & \dots \\ & p_{m1}, \dots, p_{m,n} & \Rightarrow e_m \end{array}$$

## Example: insertion sort over binary words

$t = \epsilon \mid 0 \text{ of } t \mid 1 \text{ of } t$  (*binary words*)

Algorithm to put 0's before 1's:

$$\begin{aligned} \textit{sort} = \quad \epsilon & \Rightarrow \epsilon \\ 0(x) & \Rightarrow 0(\textit{sort}(x)) \\ 1(x) & \Rightarrow \textit{ins}(\textit{sort}(x)) \end{aligned}$$

$$\begin{aligned} \textit{ins} = \quad \epsilon & \Rightarrow 1(\epsilon) \\ 0(x) & \Rightarrow 0(\textit{ins}(x)) \\ 1(x) & \Rightarrow 1(1(x)) \end{aligned}$$

## Bounded recursion on notation [BRN]

$$\begin{array}{ll} f = \epsilon, \vec{y} & \Rightarrow g(\vec{y}) \\ 0(x), \vec{y} & \Rightarrow h_0(f(x, \vec{y}), x, \vec{y}) \\ 1(x), \vec{y} & \Rightarrow h_1(f(x, \vec{y}), x, \vec{y}) \end{array}$$

with  $|f(x, \vec{y})| \leq P(|x|, |\vec{y}|)$ ,  $P$  polynomial.

## The need for the polynomial bound

- Consider first the function  $d$  doubling the size of its input:

$$\begin{aligned}d &= \epsilon && \Rightarrow \epsilon \\d(0x) &&& \Rightarrow 0(0(d(x))) \\d(1x) &&& \Rightarrow 1(1(d(x)))\end{aligned}$$

- Then consider the function  $e$

$$\begin{aligned}e &= \epsilon && \Rightarrow 1\epsilon \\i(x) &&& \Rightarrow d(e(x)) \quad i = 0, 1\end{aligned}$$

- These functions are definable by recursion on notation and the size of  $|e(x)|$  is **exponential** in  $|x|$ .

**NB** Iterating  $|x|$  times polynomial time operations can generate data whose size is not polynomial in  $|x|$ .

**Theorem** [Cobham] The set of functions computable by an algorithm in BRN are exactly those computable in polynomial time.

⇒ Straightforward execution of definitions is computable in PTIME.

⇐ Simulation of TM for polynomially many steps.

## Remarks

- Completeness results are a kind of safety check (cf. Turing completeness, logic completeness,...).
- In practice, there might be a big difference: Assembler vs. Pascal, Hilbert system vs. Natural deduction,... So there is still space for research but with a different yard stick:

good algorithmic coverage.

- Weaknesses of Cobham's theorem:
  - Restricts the programmer to [recursion on notation](#). See part 2.
  - Gives no clue on [how to find a polynomial bound on size](#). See part 3.

## Part 2: Some generalizations of Cobham's theorem

J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. Habilitation à diriger des recherches. Université de Nancy, 2000.

See also: Bellantoni-Cook, Leivant, Hofmann,...

## Lexicographic order and PSPACE

$$qbf = \phi \quad \Rightarrow \text{check}(\phi, nil)$$

$$\text{check} = v(x), l \quad \Rightarrow \text{mem}(x, l) \mid$$

$$n(\phi'), l \quad \Rightarrow \text{not}(\text{check}(\phi', l)) \mid$$

$$a(\phi', \phi''), l \quad \Rightarrow \text{and}(\text{check}(\phi', l), \text{check}(\phi'', l)) \mid$$

$$\text{all}(x, \phi'), l \quad \Rightarrow \boxed{\text{and}(\text{check}(\phi', \text{cons}(x, l)), \text{check}(\phi', l))}$$

**Prop** [Bonfante et al. PSI 01]

$\boxed{\text{PSPACE} = (\text{specialised}) \text{LPO} + \text{polynomial bounds on data size}}$

## Product order, PTIME, and memoization

$$\begin{aligned} lcs = \epsilon, y &\Rightarrow 0 \\ i(l), \epsilon &\Rightarrow 0 \\ i(l), i(l') &\Rightarrow s(lcs(l, l')) \\ i(l), j(l') &\Rightarrow \boxed{\max(lcs(i(l), l'), lcs(l, j(l')))} \quad i \neq j \end{aligned}$$

**Prop** [Marion]

$\boxed{\text{PTIME} = (\text{specialised}) \text{ PO} + \text{polynomial bounds on data size}}$

**NB** Computed values are cached. See also [Ganzinger-McAllester, ICLP 2002]

### Part 3: Synthesis of size bounds

G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In Proc. *Perspectives of System Informatics*, Springer LNCS 2244, 2001.

R. Amadio. Max-plus quasi-interpretations. In Proc. *Typed Lambda Calculi and Applications (TLCA '03)*, Springer LNCS 2701, 2003.



## Quasi-interpretation (continued)

Obvious extension of assignment  $q$  to expressions:

$$q_x = x$$

$$q_{c(e_1, \dots, e_n)} = q_c(q_{e_1}, \dots, q_{e_n})$$

$$q_{f(e_1, \dots, e_n)} = q_f(q_{e_1}, \dots, q_{e_n})$$

An assignment  $q$  is a quasi-interpretation if:

for every rule  $f(p_1, \dots, p_n) \Rightarrow e$

$q_f(q_{p_1}, \dots, q_{p_n}) \geq q_e$ , over non-negative rationals

Then we can derive:

Size of values computed by $f(v_1, \dots, v_n)$ is bound by $q_f(v_1, \dots, v_n)$
--

## Quasi-interpretations vs. interpretations

Quasi-interpretations are inspired by *polynomial simplification interpretations* for termination proofs. So what's new?

- Quasi-interpretations are *easier* to find because inequality is not strict.
- *Small* polynomials are often good enough (see next).

**NB** Polynomial interpretations that on constructors behave as quasi-interpretations provide another characterization of PTIME.

## Search space: max-plus polynomials

- We shift from the algebra  $(+, \times)$  to the algebra  $(\max, +)$ .
- Work over  $\mathbf{Q}_{\max}^+ = \mathbf{Q}^+ \cup \{-\infty\}$ .  $-\infty$  is the unit of  $\max$  and  $0$  is the unit of  $+$ .
- Distribution:  $x + \max(y, z) = \max(x + y, x + z)$ .
- Exponentiation:  $\alpha x$ .
- For a –given– degree the *synthesis problem* can be expressed as the validity of a  $\exists\forall$  Presburger formula.

**NB** We have looked for something more efficient...

## Lower and upper bounds on complexity of synthesis

The synthesis problem is:

- NP-hard, quite robustly.
- NP-complete, for multi-linear polynomials.

E.g. multi-linear polynomials in 2 indeterminates have the shape:

$$\max(a_0, x_1 + a_1, x_2 + a_2, x_1 + x_2 + a_{1,2})$$

## Part 4: Generalisation to a cooperative synchronous model

- F. Boussinot et al., Reactive Programming,  
<http://www-sop.inria.fr/mimosa/rp>.
- R. Amadio, S. Dal-Zilio. Resource control for synchronous cooperative threads. Research Report LIF 22-2004. To appear in CONCUR 2004.

## Synchronous Cooperative Threads

- System of threads interacting through shared variables.
- Filter conditions on read: channel based communication is easily simulated.
- Cooperative threads, as opposed to preemptive, no locks needed.
- Natural determinism, debugging easier.
- Synchronous execution, thread can react in the next instant, to absence of an event in current instant (several implementations available).

## Example: alarm thread

Rings on ring if no signal on sig for  $n$  instants.

$al = z \Rightarrow \text{ring} := \text{prst}.stop$

$s(y) \Rightarrow \text{read sig with}$

$\text{prst} \Rightarrow \text{next}.al(n) \mid$

$[-] \Rightarrow al(y)$

## Goals

- Instant terminates.
- With bounded resources.
- Relying on a local and incremental analysis.

## Remarks

1. Asymptotic prediction of system behaviour is desirable but clearly in conflict with local analysis.
2. In practice, dynamic check on the size of the parameters needed at the end of each instant.

## Read once condition

- Each thread can go through each read instruction at most once.
- Analysis of thread behaviours becomes **functional** in:
  - parameters at the beginning of the instant.
  - values read in the registers during the instant (**hidden parameters**).
- Condition can be easily checked by a flow analysis.
- Not too restrictive (think of synchronous algorithms).
- Without it, exponential growth is easily achieved (register as an accumulator).

## Constraints for termination and size bounds

$$f(x) = \text{yield.read } i \text{ with } l \Rightarrow f_1(\text{maxl}(l, x))$$

$$f_1(x) = \circ := x.\text{next}.f(x) \dots$$

Control points	Constraints
$(f^+(x, l), \text{yield} \dots)$	-
$(f^+(x, l), \text{read} \dots)$	-
$(f^+(x, l), f_1^+(\text{maxl}(l, x)))$	$f^+(x, l) > f_1^+(\text{maxl}(l, x))$
$(f_1^+(x), \circ := x \dots)$	$f_1^+(x) > x$
$(f_1^+(x), \text{next} \dots)$	-

## Termination

**Def.** A *reduction order*  $>$  is a well founded order on terms that is preserved by context and substitution.

**Prop.** If the termination constraints are satisfied by a reduction order then *every instant terminates*.

## Size bounds

Suppose the assignment  $q$  satisfies the *size constraints*. Then the size of the values computed by the thread  $f(\vec{v})$  during an instant is bound by

$$q_{f+(\vec{v},\vec{u})}$$

where  $\vec{u}$  are the values contained in the registers *when* they are read by the thread  $f$ .

## Combination of termination and size bounds

If the system terminates by LPO and  $q$  is polynomially bound then the computation of the system in an instant runs in PSPACE in the size of the parameters of the threads at the beginning of the instant

- A thread can read the output of another thread, but this can happen a finite number of times in an instant.
- No asymptotic bound. This needs global analysis. Here size of data has to be checked periodically.

## **Part 5: Pushing the verification at byte code level**

R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. Research Report LIF 17-2004. To appear in *Computer Science Logic* 2004.

## Motivations

- To be useful, bounds have to be *precise* and they have to be *valid* for the implemented abstract machine.
- Mobile code comes in pre-compiled form, i.e. *byte-code*.
- We should be able to perform the verifications at this level.

## CRISS approach

In CRISS, we take the a *typed assembly language* approach (cf. [Morriset et al.](#)):

- Byte code comes with [resource annotations](#). E.g. an order to check RPO termination, (max-plus) polynomials to check size bounds.
- We define [virtual machine](#) and [well-certified byte code](#).
- We prove (on paper first, in COQ later) that [well-certified byte code runs within certain bounds](#)...

## A virtual machine (for the functional fragment)

- A frame  $(f, i, v_1 \cdots v_n)$ .
- A memory  $M$  is a stack of frames.
- Byte code is organised in segments: one for each function.

## Byte code instructions

$f[i]$	Current memory	Next memory
load $k$	$M \cdot (f, i, \ell \cdot v \cdot \ell')$	$\rightarrow (f, i + 1, \ell \cdot v \cdot \ell' \cdot v) \quad  \ell  = k - 1$
branch $c \ j$	$M \cdot (f, i, \ell \cdot \mathbf{c}(v_1, \dots, v_n))$	$\rightarrow M \cdot (f, i + 1, \ell \cdot v_1 \cdots v_n)$
branch $c \ j$	$M \cdot (f, i, \ell \cdot \mathbf{d}(\dots))$	$\rightarrow M \cdot (f, j, \ell \cdot \mathbf{d}(\dots)) \quad \mathbf{c} \neq \mathbf{d}$

$f[i]$	Current memory	Next memory
build $c\ n$	$M \cdot (f, i, \ell \cdot v_1 \cdots v_n)$	$\rightarrow M \cdot (f, i + 1, \ell \cdot \mathbf{c}(v_1, \dots, v_n))$
call $g\ n$	$M \cdot (f, i, \ell \cdot v_1 \cdots v_n)$	$\rightarrow M \cdot (f, i, \ell) \cdot (g, 1, v_1 \cdots v_n)$
return	$M \cdot (g, i', \ell') \cdot (f, i, \ell \cdot v)$	$\rightarrow M \cdot (g, i' + 1, \ell' \cdot v)$
stop	$M$	$\rightarrow \epsilon$

**NB** Full machine for concurrent language has around 10 instructions.

## Example: compilation into byte code

Source code	Byte code for <i>add</i>
<i>add</i> =	1 : load 1
<i>z, y</i> $\Rightarrow$ <i>y</i>	2 : branch z 4
<i>s(x), y</i> $\Rightarrow$ <i>add(x, s(y))</i>	3 : return
	4 : branch s 9
	5 : load 2
	6 : build s 1
	7 : call <i>add</i> 2
	8 : return
	9 : stop

## Constraints generation for *add* segment

Byte code	Shape	Sub.
1 : load 1	$x_1 \cdot x_2$	$id$
2 : branch z 4	$x_1 \cdot x_2 \cdot x_1$	$id$
3 : return	$x_1 \cdot \boxed{x_2}$	$id$
4 : branch s 9	$x_1 \cdot x_2 \cdot x_1$	$id$
5 : load 2	$s(x_3) \cdot x_2 \cdot x_3$	$[s(x_3)/x_1] = \sigma$
6 : build s 1	$s(x_3) \cdot x_2 \cdot x_3 \cdot x_2$	$\sigma$
7 : call <i>add</i> 2	$s(x_3) \cdot x_2 \cdot x_3 \cdot s(x_2)$	$\sigma$
8 : return	$s(x_3) \cdot x_2 \cdot \boxed{add(x_3, s(x_2))}$	$\sigma$
9 : stop	$x_1 \cdot x_2 \cdot x_1$	$id$

$$add(x_1, x_2) > x_2 \quad \sigma(add(x_1, x_2)) = add(s(x_3), x_2) > add(x_3, s(x_2))$$

## Interpretation of the constraints

- Assume flow graph is sufficiently well structured.
- If there is an assignment  $q$  that satisfies the constraints then we can bound the size of the values in a frame as a function of the size of the values at the beginning of the computation.
- If there is a reduction order that satisfies the constraints then we can show termination of the execution.
- Again, by taking polynomially bound assignments and suitable reduction order we can give (explicit) polynomial bounds on the resources needed for the execution.

## Part 6: Some ongoing work in the CRISS project

<http://www.cmi.univ-mrs.fr/~amadio/Criss/criss.html>

1. **COQ** formalisation of byte code verification algorithms [Coupet-Delobel].
2. Shape/size analysis for optimized code [Dal Zilio-Gascon].
3. **ESTEREL**-like operators: when, watch,... [Dabrowski].
4. Termination by compilation to **Petri Nets** [Marion-Moyen].
5. **Higher-order functions** (integrate LL ideas) [Baillot-Mogbil].
6. Types for control of interference [Boudol-Castellani-Matos].